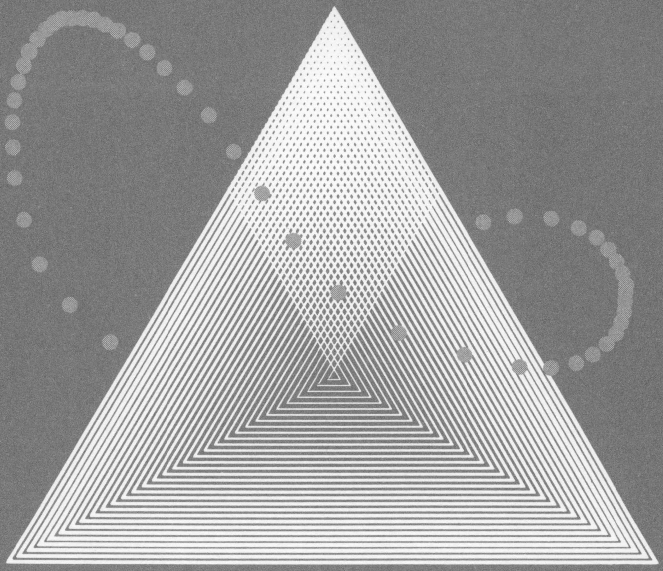
CP/Mによる
Z80マクロ・アセンブラ入門

押野崇芳――著

日刊工業新聞社



CP/Mによる Z80マクロ・アセンブラ入門

押野崇芳――著

日刊工業新聞社

はしがき

近頃，マイクロコンピュータのプログラミングに，BASIC言語やC言語をはじめとする高水準言語が多く使用されるようになった．これらの高水準言語は，BASIC言語に代表されるように，初心者にもなじみやすく，修得するのもアセンブリ言語に比較すればはるかに簡単である．しかし，一部の応用プログラムを除けば，アセンブリ言語以上に実行効率のよい言語はない．これはアセンブリ言語の場合には，使用するCPUの機能を100％引き出せるプログラムが記述できるのに対して，高水準言語から出力されるオブジェクト・プログラムには，かなり冗長な部分が含まれているからである．

このように「プログラムの実行効率」という面では，遥かに優れたアセンブリ言語であるが，残念ながら記述性の良さ，アルゴリズムを考える上での容易さを含めた「プログラム開発の効率」という面においては高水準言語の足元にもおよばない．

しかし，**マクロ・アセンブラを使用してプログラムをライブラリ化することにより，先にあげたアセンブリ言語の欠点をかなりの部分まで補うことができる**のである．

ところが，このアセンブリ言語の欠点を補えるマクロ・アセンブラに関する解説書はきわめて少なく（現在のところ，MACRO−80を本格的に解説したものは皆無である），本書で使用している“MACRO−80”というマクロ・アセンブラも，単にZ80ニーモニックを記述するためのアセンブラとして使用し，マクロ定義などのマクロ・アセンブラ本来の機能を利用しているユーザーはきわめて少ないようである．これは，著者が講師をしている（株）トリケップスや，各社の企業研修でのアセンブラ・セミナーで寄せられる質問の大部分が，このマクロ・アセンブラの使用法に関するものであることからも明白である．

そこで本書では，これまでに寄せられた質問に応える意味で，

◇マクロ・アセンブラを使用してプログラム開発を行なうメリット……………	第1章
◇MACRO−80 というマクロ・アセンブラの使用法のテクニック…………	第2章
◇CP/M でのアセンブラ・プログラム開発のためのツールの使用法………	第3章

ということを解説した．

本書を学習する場合，読者の方はアセンブラに関する理解度により，

◇マクロ・アセンブラを初めて学習する場合（入門者）

第１章（読み流す）──→ 第３章（理解する）──→ 第２章（研究してみる）

◇マクロ・アセンブラを使用したことがある場合（中級者）

第１章（検討する）──→ 第２章（確かめて理解する）

第３章（マニュアルがわりに利用する）

◇マクロ・アセンブラを実務に利用している場合（上級者）

第２章（確認してみる）──→ 第１章（参考にする）

第３章（マニュアルがわりに利用する）

という順序で学習するとよいであろう．

各章の概要は次のとおりである．

第１章は，なぜマクロ・アセンブラを利用するとアセンブリ言語によるプログラムの開発効率が向上するのかをライブラリ化という観点から解説してある．この章は，本書の理論編ともいうべきところで，内容がやや抽象的になるので，「チャチャ丸」と，「Mr. じゅげむ」という登場人物を配し，物語形式でライブラリ化の有効性をイメージとして捉えられるように漫画などもいれて解説したので気軽に読んでもらいたい．

第２章は，本書の技術的なポイントで，マクロ・アセンブラを使いこなす上で必要なアセンブラの構造とアセンブルのされ方，マクロ定義に関する命令語の使用法のテクニックなど著者が研究したことがまとめてある．残念ながら紙面の関係上，実例が不足したが，これまでマクロ・アセンブラを使用していて疑問に感じていた方への大半の質問には応えられると確信している．入門者にはやや難解かも知れないが，熟読をお勧めする．

第３章は，このマクロ・アセンブラを使用する場合に必要となるCP/M上のソフトウェア・ツールの使用法をまとめたもので，それぞれのマニュアルに解説されていることを使用例をあげて詳細に解説したものである．

コマンドの一覧表などは，実際に著者が使用してみて便利なように再編集してあるので，これらのソフトウェア・ツールを使用する時のリファレンス・マニュアルがわりに使用するのもよいであろう．

本書は，「マクロ・アセンブラ入門」ということで，マクロ・アセンブラを利用したライブラリ・プログラムの開発例は割愛したが，近い将来，本書の続編として「マクロ・アセンブラによる“ライブラリ開発応用例”」(仮題)を発表する予定である．本書は，そのための基礎知識と考えて頂ければ幸いである．

なお，執筆にあたって，本書の必要性を認め貴重な御意見をいただいた（株）トリケップスのスタッフの方々をはじめ，出版にあたって多大な御尽力をいただいた日刊工業新聞社出版局のみなさま，および，資料の整理と執筆，校正を通して精神的な援助を惜しまなかった野上晶子嬢に対し心から深謝いたします．

昭和59年9月

著　者　しるす

本書で使用しているソフトウェアに関しては，次の通りである．

◇CP/M NEC PC 8801 2W用
◇ZSID　ディジタル・リサーチ　v 1.4
◇MACRO－80関係（MACRO－80，CREF－80，LINK－80，LIB－80）
　マイクロ・ソフトMACRO－80パッケージ　v 3.44

MACRO－80関係のソフトウェアで，v 3.44以前のものには本書で解説したコマンドの中で使用できないものがあるので注意すること．

目　次

第1章　プログラム開発におけるライブラリ化の有効性

第2章　マクロ・アセンブラの使用法のテクニック

第1章

プログラム開発における ライブラリ化の有効性

[この章のポイント]

◎ なぜライブラリ化するとプログラムの開発効率がよいのか？

◎ ライブラリ化のためには，プログラムをモジュール別に開発する必要がある.

◎ ライブラリの効果的な活用には，ドキュメンテーションが重要になる.

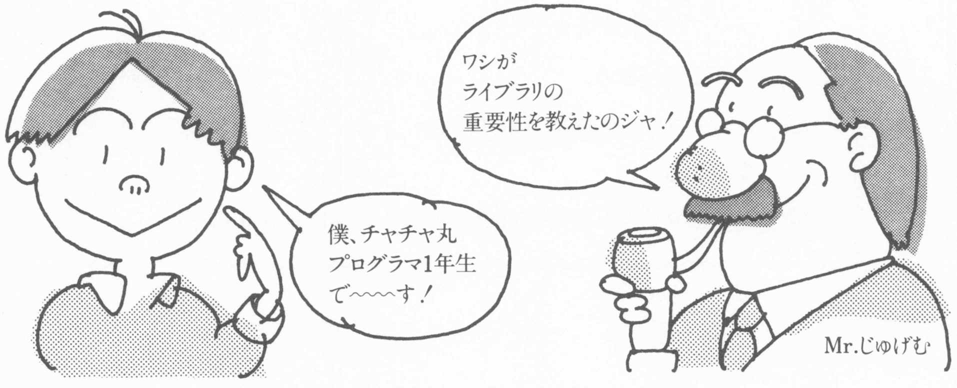
◎ ライブラリの品質管理は慎重に行なわなければならない.

I know how to use it, but
I don't know what is it!
My name is チャチャ丸

当社の財産 ⟶
チャチャ丸にも使えます！

じゅげむ　スーパーライブラリ
for CP/M program
Copyright by じゅげむ

社内重要書類　no. 10077



[チャチャ丸のプロフィール]	[Mr. じゅげむのプロフィール]
昭和59年　私立5流大学卒業 成　　績　オールC 適　　性　なし 現　　在　当社の有能なプログラマ？	この道，30年のベテラン・プログラマ．開発効率のよくないアセンブリ言語によるプログラミングにライブラリ化を導入し，新人プログラマの教育と，開発効率の向上に効果をあげている．

1.1　ライブラリ化の必要性

[レクチャー]

近年，プログラムの開発効率うんぬんということがさかんにいわれるようになってきた．これは，INSに象徴される高度情報化社会の到来により，爆発的にコンピュータのためのソフトウェアの需要が増大したために，従来の職人芸的なプログラムの開発ではとてもその需要に供給が追いつかなくなってきたからである．これはまたハードウェアとソフトウェアの進歩の比較からも論ずることができる．

図1.1は，数年前にアメリカで発表されたハードウェアの性能（価格性能比）とソフトウェアの生産性との比較データである．この図からわかるように，ハードウェアの進歩は，10年間で100倍ずつのびている．それに対して，ソフトウェアの生産性は，ほとんど横ばいといってもよいであろう．これは，ちかごろはやりのOA（オフィスオートメーション）で比較される工場の生産性の進歩と，オフィスワークの生産性の進歩とよく似ている．どちらも人手による作業の進歩は遅れているということである．すなわち，ハードウェアの生産性にはCAD（コンピュータ支援設計システム）をはじめとする機械化がその生産性の向上に大きく貢献しているが，ソフトウェアの生産性の向上には，そのような決め手となる手法が現われなかったわけである．

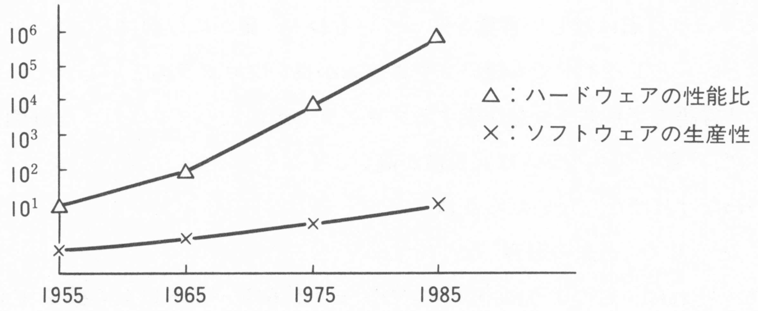


図1.1　ハードウェアの性能比とソフトウェアの生産性

ましてやマイクロコンピュータの出現により，これまでは，きわめて一部のプログラマのみが手がけてきたアセンブリ言語によるプログラム開発の必要性が高くなってきた．このアセンブリ言語によるプログラム開発というのは，生産性の向上とは全く逆の方向を示すもので，10年以上も昔の環境にもどってしまったといえる．であるから，生産性を考えるならば，このようなアセンブリ言語によるプログラム開発などということは行なわないにこしたことはない．現に16ビットCPUを使用したマイクロコンピュータ・システムでは，アセンブリ言語の代わりにC言語が使用されることが多くなった．

しかし，マイクロコンピュータ用のプログラムの場合，ハードウェアとの関係が非常に密接な

プログラムを要求されることが多いために，全くアセンブリ言語をぬきにしてこれらを行なうことはできない．そこで，この生産効率のよくないアセンブリ言語を少しでも効率よく使用する方法が必要になってくるわけである．

その方法としては，次のようなことがあげられる．

- 職人芸的なプログラムをつくらずに，だれにでもわかりやすいプログラムをつくる．
- プログラム開発は，何人かのチームで分業して行なう．
- 以前作成したプログラムをなるべく利用する．

これらのことをまとめると，プログラムをモジュール別に開発し，それらをライブラリという形で蓄積することにより，高度な技術を知らない初級プログラマでも，その力に応じた役割を開発チームの中で果たすことができ，ひいては生産効率の向上につながるということなのである．

◇

Mr.じゅげむ　チャチャ丸君，ライブラリ化の必要性がわかったかな？　当社はいち速くこのライブラリ化に取組み，社内の標準化を行なったのだよ．だから，君のようなプログラマ1年生でもすぐにプロジェクトの一員として即戦力となることができたわけだよ．

チャチャ丸　はあ！　なんとなく．しかし，いくつかわからないところがあるのですが，例えば，“だれにでもわかりやすいプログラム”というのは冗長度が高くなるのではありませんか？

Mr.じゅげむ　ほう，君は難しい言葉を知っているね！　確かに以前は，マイクロコンピュータのプログラムというと1バイトでも短いプログラムが良いプログラムといわれていた．そのために，アセンブリ言語でプログラムを作成するプログラマが他人のプログラムを評価するときには，口ぐせのように，“君のプログラムは冗長度が高いよ！”といっていたものだが，それはもはや過去の形容といわなければならないだろうね．

チャチャ丸　どうして“過去の形容”なのですか？

Mr.じゅげむ　それは，君のようにパソコンのメモリが64kバイト以上も標準実装される時代にマイクロコンピュータを使い始めた者にはわからないかもしれないが，昔は，メモリの価格というものが非常に高価で，数kバイトのメモリ実装が標準であったころには，まず第1条件はメモリの使用量をできるだけ少なくすることだったんだ．ところが，先の図（図1.1）からもわかるようにハードウェアの性能対価格比は飛躍的に向上したから，それほどメモリの使用量を気にする必要がなくなったんだよ．だから，むしろメモリの使用量に関する冗長度よりも，そのために要する開発期間のほうが問題になるわけだね．ワシも昔は，10バイトプログラムを短くするために5日間も徹夜をしたという経験があるが，それこそ開発期間の冗長度といわなければならないだろう．

チャチャ丸　なるほど，では，もうメモリの使用量などということには無関心にプログラミングすればよいのですね．

Mr.じゅげむ　そこが君の単細胞なところだな！　もちろん，基本的にはそうなのだが，メモリを多く使用するということは，すなわち処理速度が遅くなるということだ．だから，ケースバイケースでものごとは考えなければならない．特に高速のI/O機器とのインターフェースをとるような場合には，なるべく冗長度の小さいプログラムにしないとタイミングがとれなくなってしまう．そのようなときには，メモリの使用量という観点からではなく，処理速度の向上という観点から，1バイトでも短いプログラムにすることが必要なんだよ．

チャチャ丸　？？？？？　僕のようなプログラマ1年生には，その判断ができません．結局どうすればよいのですか？

Mr.じゅげむ　そのためにも，標準化を行ない，そのような部分に関しては，ライブラリの形にしておけば，ライブラリ内のテクニックはわからない君のような初心者でもそのライブラリを使用して効率の良いプログラミングができるようになるんだよ．

チャチャ丸　なるほど，でも，それではいつまでたっても僕はそのライブラリのお世話になってばかりで……，結局，僕のこの会社での存在価値はなんなのでしょうか？　昔のような職人的プログラマであれば，会社も僕の存在価値を認めて給与もあげてくれたでしょうに？？？？

Mr.じゅげむ　君は，2つの大きな誤りを犯しているね！　その1つは，今君はこのマイクロコンピュータのプログラマとしては，まだ初心者だということだ．当社では先に説明したように，標準化を行ないライブラリの構築を行なっていたから君のような初心者でも，プロジェクトの一員として早くから仕事に参加することができたが，このライブラリがなければ，まだ君は，お茶くみしかできなかったハズだよ．しかし，君はこのライブラリのおかげで，すでに何本かのプログラム開発に参加した．そこから得られたものは，単に書物から得るものにはかえられないものがあるハズだ．そして君もそのうちに，有用なモジュールを開発し，ライブラリに加えられる日がくるだろう．

今，君が使用しているライブラリも，もともとは君の先輩が残したものなんだ．

その2つ目は，“プログラマ30歳停年制”といわれることだ．昔のプログラマは，職人芸的なプログラムをよしとしてきた．そのために，チームを組んで作業を進めるということが上手ではなかった．

それゆえ，少し大きなプログラムを作成するような場合には，徹夜に徹夜を繰り返すというようなことが常識のように行なわれてきた．そのようなことを繰り返せば，当然，身体はもとより精神的にも健康を害することになる．

また，その仕事以外の知識を得る時間的余裕もなくなる．そのようなことから，30歳を過ぎればもう役立たずの人材となってしまったんだよ．ワシの頭がうすくなったのもそのせいじゃよ．これからの時代に生きるプログラマは，自分にしかできない仕事などというものをつくってはいけない．自分の作成したモジュールで汎用性のあるものは，どんどんライブラリ化していき，メンバーの総合的な開発力を向上させることがすなわち自分の能力を向上させることにつながるん

だよ．

チャチャ丸　なるほど，なんとなくわかったような気がします．ところで，これまでの話の中で“モジュール”という言葉が何度となく出てきましたが，“モジュール”と“ライブラリ”はどのような関係があるのですか？

Mr.じゅげむ　いい質問だね．では，次にこの“モジュール”ということを考えてみよう．

1.2　ライブラリ化のためのモジュール開発

1.2.1　モジュール開発の利点

［レクチャー］

複雑なプログラムとはどのようなものであろうか？　最も簡単にいえることは，ステップ数の多いプログラムは難しいといってもよいであろう．例えば，内容的には非常に高度なプログラムであっても，それが数行程度のプログラムであれば，その処理内容を理解するのにそれほどの労力を必要としない．ところがその逆に，内容は簡単なものであっても，それがプリント用紙に数十枚にも及ぶものであれば，その処理内容を理解することは容易なことではない．ある調査によると，120 種類の PL/I プログラムを調べた結果，1 つのプログラム中で，ステートメント数の平均は 853 ステートメントになり，384 の変数が 1195 回使用されていたそうである．このようなプログラムの処理内容を理解するのは大変な作業であろう．だいいち，その中で使用されている変数の意味を記憶することすら並大抵のことではない．このようなプログラムを簡単にするためには，どのようにすればよいかというと，大きいプログラム（ステップ数が多い）を小さいプログラムにかえればよいわけである．しかし，プログラムの仕様(処理内容)がかわらない限り，100 ステップのプログラムが，10 ステップになるというようなことはありえない．ここでいうところのプログラムを小さくするというのは，大きなプログラムをいくつかの小さな部分に分割するということである．この分割された 1 つの部分を“モジュール”と呼んでいる．書籍に例えれば，目次に表わされる小見出しのようなものである．このようにモジュール化することによって，大きいプログラムを小さなプログラムの集合にすることができる．すなわち，複雑なプログラムを簡単なプログラムの集合にかえることができるわけである．また，モジュール化することにより，1 つの大きなプログラムをモジュールごとに分けて，複数の開発スタッフにより開発を行なうことができる．もう 1 つ重要なことは，このようにして開発したモジュールは，そのプログラムだけでなく，それ以降に開発するプログラムのモジュールとしても再利用可能なものが多いということである．この代表的なものとしては，CRT ディスプレイへの表示や，キーボードなどからの入力をはじめ，よく使用される関数などは，そのパラメータの設定を工夫することによって，一度作成しておけば，どのようなプログラムにも共通して使用することができるようになる．

すなわち，このような汎用モジュールをライブラリとして利用するわけである．これらをまとめると，次のようになる．

1. 大きなプログラムを，小さなプログラムの集合にできる．すなわち，複雑なプログラムが簡単なプログラムになる．　　──→　作業の簡単化
2. 1 つの大きなプログラムが，いくつかのモジュールになることにより，作業の分担

が行なえるようになる. ⟶ 開発期間の短縮
3. あるプログラム用に作成したプログラム(モジュール)を，他のプログラムのモジュールとして再利用することができる. ⟶ 開発効率の向上

Mr.じゅげむ チャチャ丸君，プログラムをモジュールに分割して開発することの利点が理解できたかね？

チャチャ丸 はい，よくわかりました．でもまたいくつか質問があるのですが，その1つ目は，大きなプログラムをモジュールという小さなプログラムに分解すれば考え方が簡単になるということはわかるのですが，そのモジュールに分解する方法がよくわかりません．何かルールがあるのですか？

Mr.じゅげむ これはよい質問だ！ それにはまずモジュールの定義をもう少し明確にしてみよう．マイヤーズという人がモジュールを次のように定義している.

[マイヤーズによるモジュールの定義]

1. 複数の命令文が語いとしてまとまっており，境界識別子(START，END など)で区切られ，固有の名前が付いている.
2. 別々にコンパイルすることができる.
3. 他のモジュールから呼び出すことができる.

チャチャ丸 これは何ですか？ まるでアセンブリ言語の世界とは，ほど遠い言葉が並んでいますね！

Mr.じゅげむ そのとおり，このマイヤーズの定義というのは“構造化プログラミング理論”の中で述べられていることで，このプログラム理論を厳密に反映できるプログラム言語は，現在のところPASCALくらいしかないといわれているんだよ.

チャチャ丸 じゃあ，アセンブリ言語なんかじゃどうしようもないじゃないですか！

Mr.じゅげむ いやいや，それは早計というものだよ．アセンブラの中にもSMAL 80と呼ばれる高水準言語に近いアセンブラもあるし，これから君に解説しようとしているマクロアセンブラを使用しても，かなりこの構造化プログラミングの思想を導入することはできるのだよ．例えば，マイヤーズの定義の2番目の“コンパイル”という言葉を“アセンブル”という言葉にかえればこれに関しては問題がなくなる．また，3番目の定義は，“サブルーチン”のことだと思えばよいわけだ．また1番目の定義は，16ビットCPU用のアセンブラであるマイクロソフト社の“MACRO 86”などには，“プロシジャー(PROC)”という擬似命令が用意されている．これを使用すればある程度までこの定義を満足することができるわけだよ.

チャチャ丸 なるほど，頭を柔軟にして考えなければいけないということですね．しかし，この定義だけではやはりモジュールに分解するときの決め手とはなりませんね．もう少し具体的に


◉開発期間の短縮
納期が
楽になる
◉作業の簡単化
休みが
増えるナア～
◉開発効率の向上
マネージメントが
簡単になるナ！
これらが
モジュール開発の
利点じゃよ！

モジュール化の方法を教えてください．

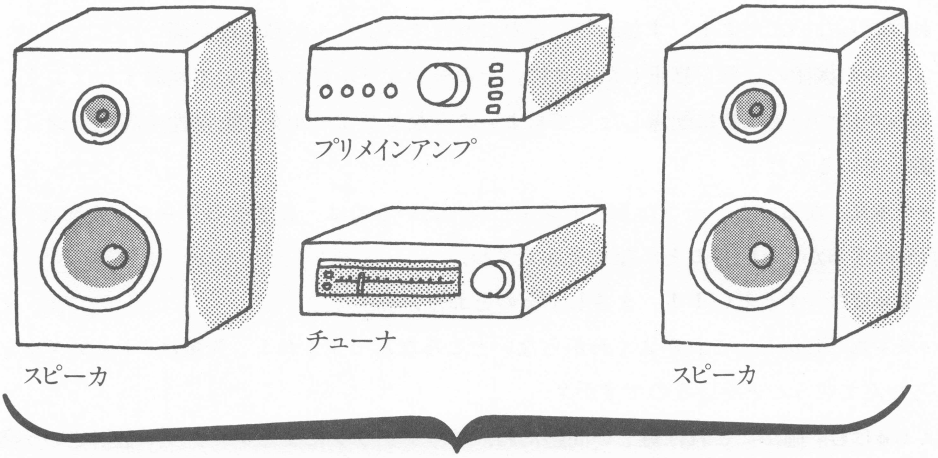
Mr.じゅげむ　このマイヤーズの定義は基本原則のようなものだから，これを実際のプログラム設計に応用するには，もう少しモジュールについて考えてみることが必要になるね．すなわち，もとのプログラムをモジュールとして，どのように分割するかが問題になるわけだ．もし，この分割の方法がよくなければ，たとえプログラムを小さく分割しても，わかりやすい構造とはならないし，また，そのようなモジュールをライブラリとして登録しておいても，他のプログラムで利用するというような機会はないだろうね．では，どのような分割の方法がよいかということだが，数学の方程式のように“必ずこうすればよい”というものはない．しいて言葉にするならば“モジュールの独立性”を十分に考慮して分割しなさいということだ．

チャチャ丸　またまた，話が難しくなりましたね．モジュールの独立性とはいったい何なんですか？

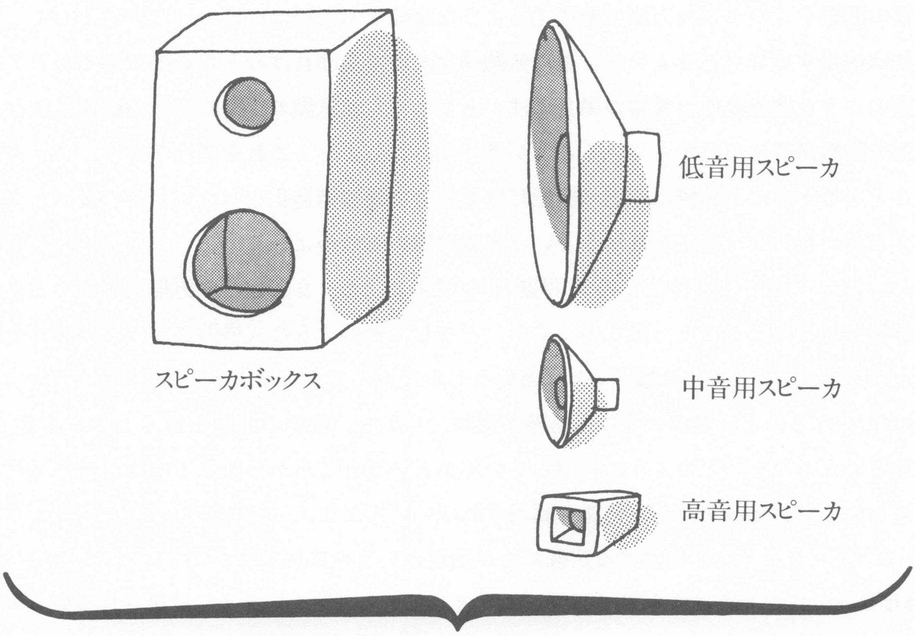
Mr.じゅげむ　では，これを一般的なことがらで説明してみよう．このごろ，ラジカセというのが若者の間で流行しているね．スピーカとチューナとカセットテレコとアンプが１つのケースに納まったもので，どこへでも移動できるステレオシステムのようなものだね．これはシステムとして使用する場合には，きわめてコンパクトで便利なものだが，もし，その中のスピーカをより高性能なものにかえたいと思っても，その部分だけを交換することはできないね．ところがコンポーネントステレオと呼ばれるものは，その一部分だけでも容易に交換ができるというわけだ．これをプログラミングにあてはめてみると，ラジカセはモジュール化されていないプログラムといえるし，コンポーネントステレオはモジュール化がなされたプログラムということがいえるんだよ．しかし，このコンポーネントステレオでも安価なものは，本体としてアンプ，チューナ，カセットが一体になっていて，それにスピーカを接続するというようなものもある．これらは，あまりよいモジュール化とはいえない．なぜならば，本体の機能が複合化されているから，それぞれの機能を独立に交換するというようなことができないからだ．この例からもわかるように，１つのモジュールは，１つの機能になっていなければならない．これが“モジュールの独立性”ということになるわけだ．ところが，この独立性ということに関しても，あまりこだわりすぎて特定の機能をいくつかのモジュールに一部分ずつ分散させてしまうと，かえって複雑さが増加してしまうこともあるんだよ．

チャチャ丸　それはどうしてですか？　機能をできるだけ細分化したほうがモジュールの独立性は高くなるのではないですか？

Mr.じゅげむ　君のような初心者が，必ず犯す誤りがそこにあるんだ．この問題に関しても，先のステレオの例で考えてみよう．この中でスピーカについて考えてみると，高級なスピーカシステムでは，スピーカボックスの中に，低音用，中音用，高音用という３種類のスピーカが入っていることが多い．これを，それぞれ１つずつのボックスにモジュール化してしまったとしよう．これはよいモジュール化といえるだろうか？　音というものは“低・中・高”のバランスがとれていて，はじめて良い音に聴こえるものだから，このようなモジュールに分けてしまうと，その



効率的なモジュール化



ここまで分けると、
よいモジュールとはいえない。
これはパーツ[部品]である。

1つを変更することによって他のモジュールをも変更しなければ音のバランスが狂ってしまう．つまり“細分化が必ずしもモジュールの独立性を高めることにはならない”ということに注意しなければならないんだよ！　すなわち，どのようなモジュールが独立性が高いかというと，そのモジュールの機能の一部を修正したときに，そのモジュールだけについて検討すればよく，他のモジュールについては特に配慮しなくてもよいようなモジュールを“独立性の高いモジュール”と定義しているんだよ．

チャチャ丸　なるほど，そうなるとモジュール化というのは，まずそのプログラムをどのように“機能分けするか”ということになりそうですね．

Mr.じゅげむ　するどい！！　まさしくそのとおり．

チャチャ丸　しかし，これがよくわからないところなんですよね！　先輩は“トップダウン”で考えろなんていうことをいうのですが？

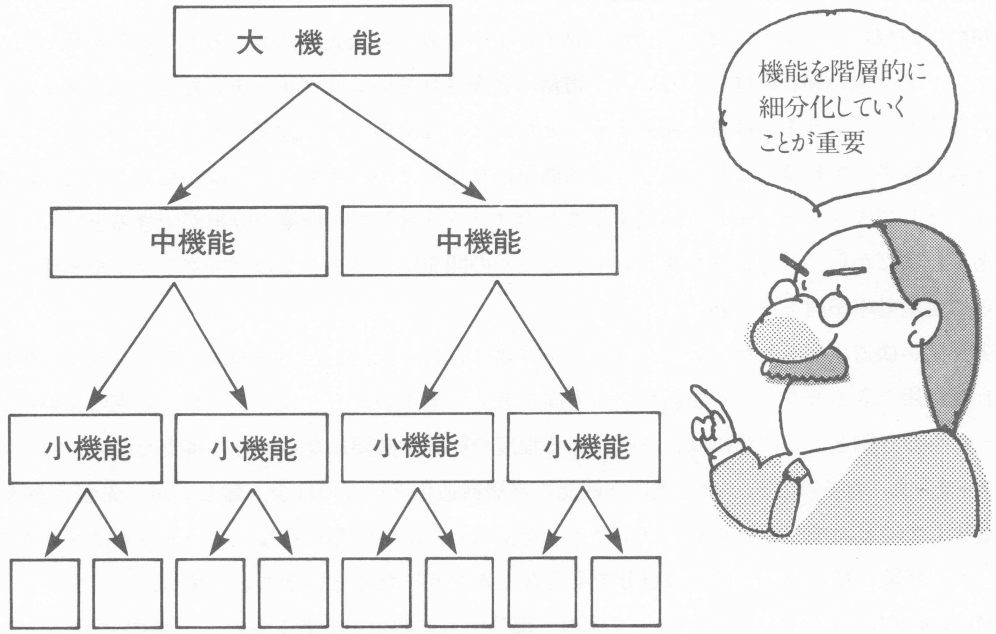
Mr.じゅげむ　確かにこれは難しい問題だね．トップダウンによるプログラム設計という発想は，そこにAという問題が与えられたときに，そのAはどのような機能からなっているかを考え，次に，そこで得られた機能をさらに細分化していくという方法で“機能の階層構造化”と呼んでいるが，これはプログラミングの経験がないとなかなか難しい問題だ．これを修得するためには，つねに日常の問題をこのような方法で整理するような訓練をしていないといけない．そして，本来この方法は使用するプログラミング言語に無関係なものでなければならないのであるが，アセンブリ言語のような機能の低い言語の場合には，どうしても無意識のうちに言語の機能に依存するような形で階層構造化を行なっていることが多くなってしまう．これを避けるには，プログラムということを考えずに，純粋に問題の機能だけを考えて階層構造化を行ない，その後に，プログラミング言語に合わせた補正を行なうという方法を用いるのがよい．

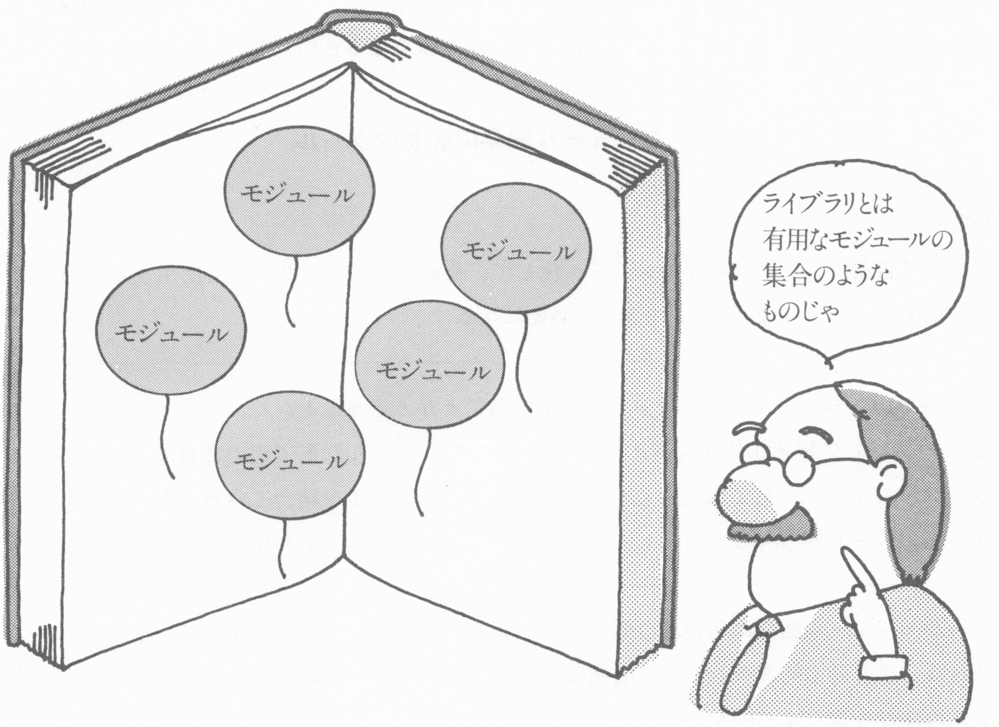
チャチャ丸　そこの補正というところが問題なんですよ．ちょうど数学の方程式を解くときのように，解答を見ればなるほどと思うのですが，どうしてそのような式展開になったのか，そのプロセスとなるとまさに『あんた頭がいい』というしかないんですよね！

Mr.じゅげむ　なるほど，君のいうことも確かだね．しかし，数学の問題を解くときにも定義だとか定理というものを，どのように利用していくかというところから始まるのだから，プログラミングにおいても，そのあたりの発想を利用すればいいんだよ．

チャチャ丸　プログラミングにおける定義だとか定理というのは何ですか？

Mr.じゅげむ　これこそまさに本書で解説しようとしている“ライブラリに登録されているモジュール”ということになるわけだよ．これらのモジュールは，その使用方法さえ理解していれば，その処理がどのような方法によって行なわれているかを知らなくても，正しい入力パラメータを与えれば正しい結果を得ることができる．すなわち，すでに証明されている定理は，その証明の方法を知らなくても方程式の解法には利用できるのと同じことだよ．一般的に定理を証明することは難しい．そこで，問題を解くたびにこれを証明するのではなく，必要となることがらを定理という形で既知の事実として使用するのと，モジュールをライブラリ化しておいてこれを必要な


大　機　能
中機能
中機能
小機能
小機能
小機能
小機能
機能を階層的に
細分化していく
ことが重要


モジュール
モジュール
モジュール
モジュール
モジュール
ライブラリとは
有用なモジュールの
集合のような
ものじゃ

ときに使用するのとは，同じ論理となるわけだな．

チャチャ丸　では，最初の定理の証明はどのようにして行なうのですか？

Mr.じゅげむ　これには，2つの方法がある．1つは数学の公式集にあたるものを利用する方法だ．プログラムの場合には，いろいろな書籍に発表されているサブルーチンなどを利用するのが便利だろうな．もう1つは自ら開発することだ．これはそのときは大変かもしれないが，ライブラリに登録さえすれば，それ以降の作業は簡単になるのだから多少の苦労はいたしかたないだろうな．そしてモジュール化して開発することのメリットとして，作業の分担ができるということがあるわけだから，このような新しいモジュールの開発は，ベテランのプログラマが担当すれば，さらに生産効率が向上するわけだな．

少々話が横道にそれたが，プログラミング言語に合わせた補正というのは，ライブラリに登録された利用できるモジュールを活用する方向と考えてもよいだろうね．しかし，効率のよいモジュールへの分割という作業には，やはりある程度の経験が必要になることは事実だね．

チャチャ丸　なるほどね，今の僕には機能の階層構造化というのは少々難しいが，先輩が階層構造化してくれたモジュールを，ライブラリを使用しながら開発を行なっているうちにプログラミングの経験が積まれ，やがては自分でも行なえるようになるということですね．

Mr.じゅげむ　そのとおり．プログラムの一部分でも，頭の中で考えるのと，実際にプログラミングするのとでは，そこから得られるものには大きな相違がある．モジュール開発の場合には君のような新米プログラマでも，その能力にあった仕事を通して，このプログラミングの経験というものが得られるから，プログラミング能力の上達も速くなるということだね．

1.2.2　モジュール開発に必要なツール

[レクチャー]

アセンブリ言語のプログラムをモジュール化して，それをライブラリとして使用するためには，次の条件を満足するような“アセンブラ”が必要になる．

1. 個々のモジュールを別々にアセンブルすることができること．
2. オブジェクトプログラムのアドレスは，リロケータブルな形で出力されること．
3. オブジェクトの形で出力されたモジュールとモジュールを結合できること．
4. モジュールをライブラリとして管理するライブラリマネジャが用意されていること．

これらがモジュール開発に必要な最低の条件となる．さらに“マクロ展開”が可能なアセンブラであれば，より効率的なモジュール開発によるライブラリ化を推進することができる．

これらの条件を満たすCP/M上のアセンブラとしては，

1）マイクロソフト社の“MACRO-80システム”

2）ディジタルリサーチ社の“RMACシステム”

などが一般的である．ディジタルリサーチ社の“MAC”はCP/Mの“BIOS（Basic Input Output System）”のソースプログラムをアセンブルするためのアセンブラとして使用しているユーザーも多いが，このアセンブラは，マクロアセンブラとしてのマクロ定義の機能は優れているが，アブソリュートアセンブラであるためにモジュールとモジュールの結合ができない．そのため，このライブラリ化という目的のためには，このMACを利用することはできない．MACRO-80を使用するか，RMACを使用するかはユーザーの自由であるが，マイクロソフト社のコンパイラBASIC（BASCOM）などの高水準言語とのリンクを考えると，MACRO-80を使用するほうがよいであろう．

このMACRO-80システムには，

1）M80．COM（マクロアセンブラ）

2）L80．COM（リンクローダ）

3）LIB．COM（ライブラリマネジャ）

4）CREF80．COM（クロスリファレンス・ユーティリティ）

の4つのプログラムがパッケージの中に含まれている．このリロケータブル・マクロアセンブラはCP/Mに標準で付属してくる“ASM”などとは比較にならないほど使用法が複雑に見えるが，これはリンクローダとの関係によるもので，それぞれの機能と両者の関係，オプションのスイッチの使用法をよく理解していれば，けっして難しいものではない．それどころか，クロスリファレンス・ユーティリティを使用すれば，変数の使用状況を知ることもできるし，また，SID，ZSIDなどのシンボリック・デバッガを使用すれば，はるかに効率のよいデバッグを行なうことができる．

（A）リロケータブルオブジェクト・プログラムとは？

チャチャ丸　個々のモジュールを別々にアセンブルするというのは，マイヤーズの理論からよくわかるのですが，“オブジェクトプログラムのアドレスはリロケータブルであること”というのはどういうことですか？　オブジェクトプログラムというのは，機械語のプログラムのことでしょ．ということは，当然，アドレスが決定されるのではないですか？

Mr.じゅげむ　確かにASMやMACのようなアブソリュートアセンブラのオブジェクトプログラムというのは，メモリイメージのオブジェクトプログラムが，HEX形式という形で出力されるわけだね．君のいっているオブジェクトプログラムというのは，このメモリイメージのHEXファイルのことだろう．

チャチャ丸　HEXファイルであることはまちがいありませんが，“メモリイメージ”とはどういうことですか？

Mr.じゅげむ　なるほど，ではこの“メモリイメージ・ファイル”ということから説明しよう．これは，その名前が示すとおり，メモリに展開される機械語のコードをそのままファイルにした

```
                ;     ****  Example   No.1  ****

0100                  org     100H

0100 78        START:mov      A,B
0101 C30501           jmp     EXIT
0104 00               nop
0105 C30000    EXIT: jmp      0

0108                  end
```

《EX1.HEXの内容》

```
:0801000078C3050100C30000F3      ← アミをかけた部分がオブジェクト・コード
:0000000000
```

《EX1.COMのダンプ・リスト》

```
0000  78 C3 05 01 00 C3 00 00 00 00 00 00 00 00 00 00
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

図1.2(a) ASMによるアセンブル例

```
     MACRO-80 3.44          09-Dec-81        PAGE     1

                                   ;     ****  Example   No.2  ****

0000'   78                       START:mov       A,B
0001'   C3 0005'                       jmp       EXIT
0004'   00                             nop
0005'   C3 0000                  EXIT: jmp       0

                                       end

     MACRO-80 3.44          09-Dec-81        PAGE     S

Macros:

Symbols:
0005'       EXIT            0000'   START

No Fatal error(s)
```

《EX2.RELのダンプ・リスト》

```
0000  84 D1 56 0C A5 00 00 13 50 80 03 C3 0E 82 80 00
0010  18 60 00 04 E0 00 00 9E 1A 00 00 00 00 00 00 00
0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

図1.2(b)　リロケータブル・マクロアセンブラでアセンブルしたもの

ものをいうわけだ．**図 1.2** は，簡単なプログラムを ASM を使用してアセンブルして得られたアセンブルリストと HEX ファイル，そして，この HEX ファイルを LOAD コマンドを用いて，実行ファイル（COM ファイル）に変換したものを示してある．これを見ると，ASM から得られたオブジェクトファイル，すなわち，HEX ファイルの内容と，実行ファイルの内容は，HEX ファイルのフォーマット記述部分を除けば，全く同じものであることがわかるだろう．この実行ファイルは，そのままメモリに展開されるわけだ．このように，メモリに展開される情報と同じ内容をもったものを“メモリイメージ•ファイル”というんだよ．

チャチャ丸　メモリに展開されるのと同じものをメモリイメージ•ファイルというのですか．ということは，リロケータブルアセンブラから出力されるオブジェクトプログラムはどのようになっているのですか？

Mr. じゅげむ　これは少々難しいが，最も簡単に表現すると，プログラムの開始アドレスを，0 番地からにしたものと考えればよい．このようにすることによって，実際のアドレスに変換するときには，実際のアドレスをバイアスとして加えることにより，メモリ空間のいずれの部分へも移動が可能なプログラムになるわけだ．それゆえ，

“リロケータブル ⟶ 再配置可能”

という名前が付いているわけだよ．

チャチャ丸　なるほど，そうすると図 1.2 の例では，開始アドレスが 100 番地ですからそれに伴い“EXIT 番地”も 105 番地になっていたわけですが，それが，5 番地からになるということですね．そしてその情報を HEX フォーマットにしたものがリロケータブルアセンブラから出力されるオブジェクトプログラムということになるのですね．

Mr. じゅげむ　そのとおり．といいたいところだが少し違うんだな！　このリロケータブルアセンブラから出力されるファイルは，HEX フォーマットとはならずに，“ビットストリーム”と呼ばれる特殊なフォーマットをもつファイルになるんだよ．それともう 1 つ重要なことは，外部参照記号と，共通参照記号という情報も記録しているんだよ．

チャチャ丸　外部参照記号とか共通参照記号というのは，何のことですか？

Mr. じゅげむ　これは，**図 1.3** を見るとよくわかるが，外部参照記号というのは，そのモジュール以外のモジュールで定義されているシンボル（名前）のことで，そのモジュールをアセンブルしたときには，その内部には定義されていないので，仮の値が入れられるわけだ．図 1.3 では“moj 1”と“moj 2”がこれにあたるわけだね．すなわち，“CALL moj 1”と“CALL moj 2”の部分の機械語の部分を見ると“CD 0000*”となっている．この“*”のマークが付いている部分については，『このモジュール以外で定義されているから，リンクする際に決定せよ』ということになるわけだ．これが外部参照記号用の情報ということになり，この情報は，オブジェクトコードとは別に記録されることになる．では，“共通参照記号”はどうなるかということだが，これは，図 1.3 では，“CCP”というラベルがこれに相当するわけだが，これは，このラベルを他のモジュールでも使用しているということなんだ．これを記録したものが共通参照記号用の情報ということ

```
        MACRO-80 3.44          09-Dec-81        PAGE    1

                               ;     ****  Example  No.3  ****

                                     .Z80

                               PUBLIC           CCP          ← 共通参照記号宣言
                               EXTRN            MOJ1,MOJ2    ← 外部参照記号宣言

0000                           CCP    EQU       0

0000'   3E 0A                  START:ld         A,10
0002'   CD 0000*  ←┐                 call       MOJ1
0005'   3E 14      │外部参照マーク    ld         A,20
0007'   CD 0000*  ←┘                 call       MOJ2
000A'   C3 0000                      jp         CCP

                                     end

        MACRO-80 3.44          09-Dec-81        PAGE    S

Macros:                              外部参照マーク

Symbols:
0000I  ←┐      CCP             0003*    MOJ1            0008*    MOJ2
0000'   │      START
        └─  共通参照マーク
No Fatal error(s)
```

図 1.3　共通参照記号と外部参照記号を使用したプログラム

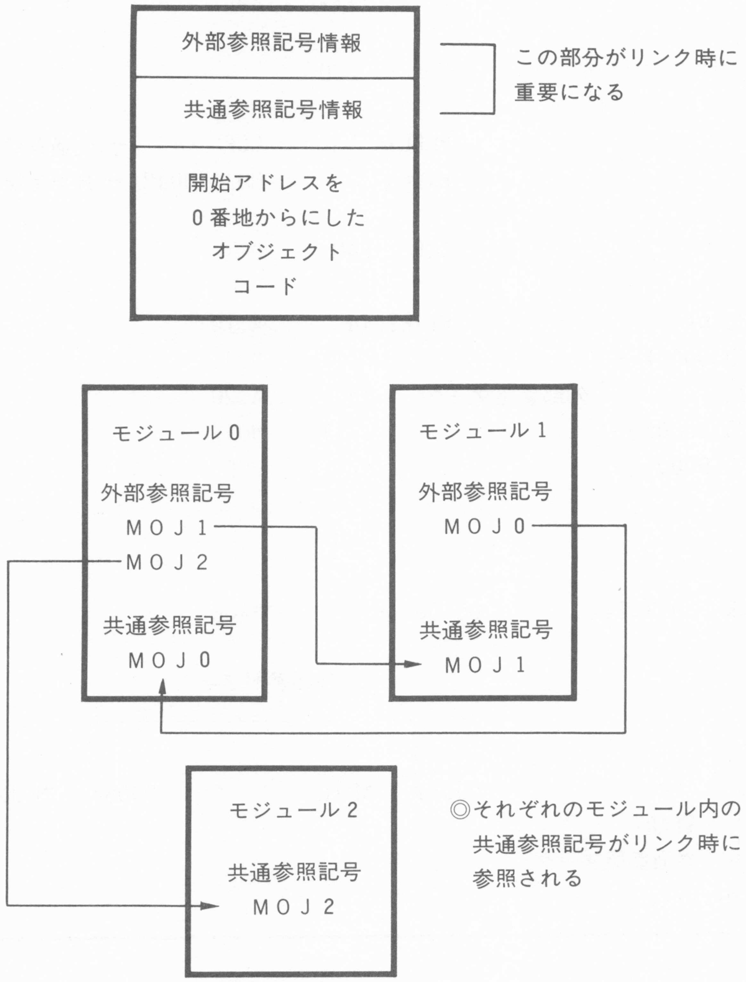


図1.4　リロケータブルオブジェクト•プログラムの内容

になり，これも外部参照記号情報と同様に，オブジェクトコードとは別に記録されることになっているんだよ．

チャチャ丸　ということは，“外部参照記号”として使用されているラベルは，そのラベルを定義しているモジュール内で必ず“共通参照記号”として定義されているということになりますね．もしこの外部参照記号が，他のモジュールで共通参照記号として定義されていなければどうなりますか？　アセンブル時にエラーメッセージが出力されるのですか？

Mr.じゅげむ　いやいや，アセンブラでは，外部参照記号はその他のモジュールで定義されているものと見なしてアセンブルしてしまうよ．なぜならば，アセンブルをする時点では，外部のモジュールに関する情報はないのだから，チェックのしようがないだろう．ちょうど，ボーナスがあると信じて働いているサラリーマンと同じだよ．しかし，ボーナスを出せるか出せないかは，経営者の腕しだいということになる．この外部参照記号の場合も同様で，プログラムを設計するものが管理しなければならないのだよ．

チャチャ丸　ボーナスが出ないというのは，いつわかるのですか？

Mr.じゅげむ　それは，ボーナスの支給日だよ！　つまり，アセンブラ株式会社としては，リンクローダで，それぞれのモジュールをリンクするときにエラーメッセージとして出力されるわけだ．話が横道にそれたが，このリロケータブルオブジェクト•プログラムの内容を図にすると図**1.4**のようになり，ASMのようなアブソリュートアセンブラから出力される“メモリイメージ•ファイル”とは全く異なる内容のものとなっているわけだね．これがリロケータブルオブジェクト•プログラムの正体ということさ．

チャチャ丸　なるほど．リロケータブルだからモジュールごとに開発しても，その後にモジュール同士を結合して1つのプログラムにすることができるのですね．

Mr.じゅげむ　そのとおり．そのモジュールの中で汎用性のあるものをまとめたものが“ライブラリ”ということになるわけだよ．

(B)　リンクローダとライブラリマネジャとは？

チャチャ丸　では，このリロケータブルオブジェクト•プログラムはどのようにしてメモリイメージ•ファイルに変換されるのですか？

Mr.じゅげむ　その働きをするのが“リンクローダ”と呼ばれるプログラムになるわけだ．このリンクローダというのは，CP/Mに付属してくる“LOAD．COM”などと比較するとはるかに複雑な機能をもっているのだが，そのポイントを図示すると図**1.5**のようになる．すなわち，

1. 開始アドレスを決定して，指定されたリロケータブルオブジェクト•プログラムをアドレスを整合しながらリンク(結合)する．
2. 外部参照記号をさがして，その値を決定する．
3. 実行ファイル(COMファイル)を作成する．

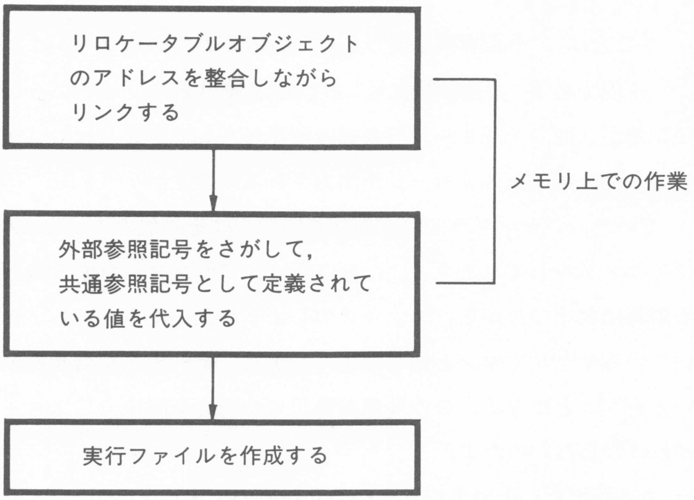


図 1.5　リンクローダの機能

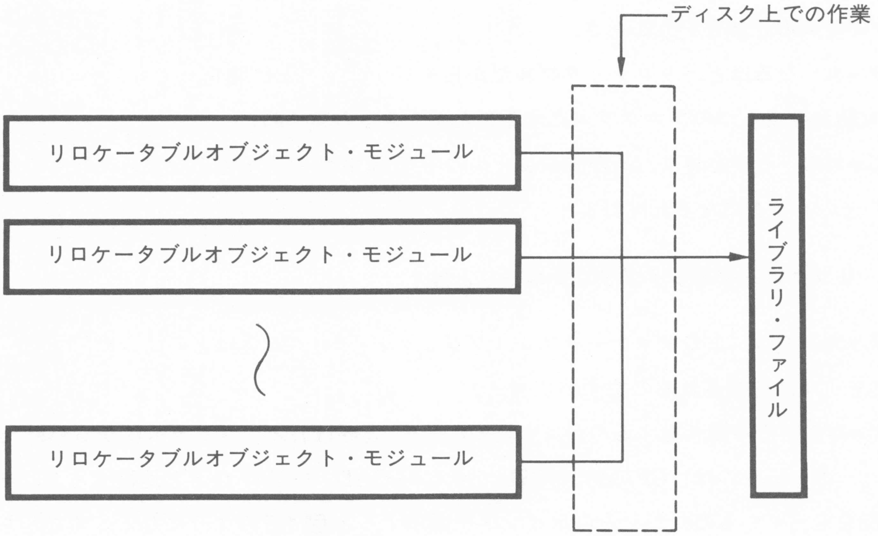


図 1.6　ライブラリマネジャの機能

この3つの作業がリンクローダのおもな機能になるわけだが，それ以外にもそのリンクローダによっていくつかの機能が盛り込まれているんだよ．

MACRO-80のリンクローダである“L80”に関しては，第3章にその使用法と機能が解説してあるからこれを参照してくれたまえ．

チャチャ丸　なるほど，そうすると，このリロケータブルアセンブラというのは，リンクローダを通すことによって，僕たちが見なれているオブジェクトプログラムが得られるわけなのですね．

Mr.じゅげむ　まあそう考えてもよいだろうね．もう少し表現をかえると，リロケータブルアセンブラに対する“LOADコマンド”のようなものと思えばまちがいないね．

チャチャ丸　次に“ライブラリマネジャ”というのはどのようなものなのですか？

Mr.じゅげむ　これはその名前が示すとおりの“ライブラリを管理するプログラム”ということだよ．つまりリロケータブルオブジェクト・ファイルの形をしたモジュールはリンクローダを使用することによってメインプログラムに結合することができるが，このモジュールの数が多くなると，その管理がきわめてメンドウになる．つまり，どのモジュールと，どのモジュールが必要だったのかを人間が管理するのでは，きわめて大変だからそれをプログラムでやってしまおうというのが，このライブラリマネジャの機能（図1.6）になるわけだ．

これも，第3章にその使用法と機能が解説してあるからそちらを参照してくれたまえ．

チャチャ丸　図書館の貸出し係のおねえさんと思えばよいですか？

Mr.じゅげむ　そのとおり．優秀なおねえさんだよ！！！！

1．3　ライブラリ開発に必要なドキュメンテーション

[レクチャー]

ライブラリ開発の第1の目的は，プログラム開発の効率化にあるわけであるから，このライブラリの運用は効率よく行なわれなければならない．そのためには，そのライブラリの機能がだれにでも容易に理解できるようなものになっていなければならない．そのためにドキュメンテーションが重要な役割を果たすようになるわけであるが，このライブラリのためのドキュメンテーションは，一般のプログラム開発のためのものとは少し性格が異なり，

“第1に使用方法を明確にする”

ということが重要になる．極端な表現をすると，その内部のアルゴリズムに関しては全くわからない人間でも，そのドキュメンテーションを見ることによって，それを使用することができるような記述がなされていなければならない．これには，

“入力条件と出力条件”

はもとより，

“べからず条件”

が重要になる．“べからず条件”とは，その名前が示すように，『これこれを行なってはいけない』ということを明確にしておくことである．これは，そのライブラリのモジュールを作成した開発者にとってはあたりまえの禁止条件であっても，それを使用する者のレベルは，自分のそれとは一致しないのだということをよく肝に命じておくことである．これは，わかっていても，つい時間的な制限から『これくらいは，あたりまえのこと』として省略してしまうことが多いが，意外とこの“べからず条件”によりバグが発生しているということが多いものである．特に，複数のメンバーで1つのプログラムを開発するような場合には，自分の常識が，他人の常識でもあると考えてはならない．このようなことを念頭においたドキュメンテーションを行なうことが，どのような技術的な問題にも先行して考えられなければならない．

このようなことから，ライブラリ開発に必要なドキュメンテーションとしては，以下のようなものが必要となる．

1. ライブラリ使用者用マニュアル
2. ライブラリ内モジュール使用マニュアル
3. ライブラリ内モジュール設計書

これらの詳細に関しては，使用する言語，開発対象となる分野により異なるので，ここでは，これ以上の解説は省略する．

◇　ドキュメンテーションにはOA機器を！

Mr.じゅげむ　このドキュメンテーションという作業は，なかなか口でいうほど簡単な作業じゃないんだよ．そこで，これを効率よく行なうためには，社内で十分に検討を行ない，それに従った標準化ということを行ない，なるべく簡単に必要事項のみを記入すればよいような用紙を作成しておくとよいかもしれないね．このごろは日本語ワードプロセッサが低価格であるからこれを利用するのもよいかもしれないね．

チャチャ丸　当社の場合には，業務日報から，プログラムのドキュメンテーションまですべてがワードプロセッサへの入力によって管理されていますね．最初は，このワードプロセッサの使用方法をおぼえるのが少々おっくうでしたが，慣れてしまえば，さて日報の用紙はどこだ，ドキュメンテーション用の用紙はどこだというように，いちいち引出しをひっくりかえすこともなくて便利ですね．

Mr.じゅげむ　そのとおり．こちらもチャチャ丸君の作業状況を知るのに，いちいち日報をさがさなくても，ワードプロセッサのキーをたたくだけで簡単に知ることができて，きわめて便利なものだよ．

チャチャ丸　これらのフォーマットはやはり社内で検討を繰り返して決めたのでしょうね．プログラムのドキュメンテーションなんてとても難しくて，僕なんかにはとてもできないと思っていたけれど，当社のシステムのようになっていると，質問に答えるだけでドキュメンテーションができてしまうのだから簡単ですね．

Mr.じゅげむ　このようなシステムをつくる場合には，その業務に最適なフォーマットを十分に検討するということがなによりも大切なことになるんだよ．他社の借りものではけっして成功することはない．当社でも，このシステムを完成させるまでには，ずいぶんと日数もかかったが，これを完成させたおかげで，当社では，君のようなけっして優秀とはいえない新入社員でも，即戦力となれるのだからね！

チャチャ丸　！！！！！！

1.4 ライブラリの品質管理

[レクチャー]

ライブラリ化によるプログラム開発の場合に重要なことは,

“ライブラリの内容は保証されなければならない”

ということである．もちろん，入力条件や出力条件を誤って使用した場合には，責任はとれないが，正しい使用法のもとでは必ず正しい結果が得られるという保証が必要になる．これは，ライブラリの内容は，数学の公式のような形で扱うのであるからあたりまえのことといえよう．そのためには，この“ライブラリの品質管理”ということを厳格に行なうことが重要である．

このライブラリの品質管理には，

1. ライブラリ自体の品質管理
2. ライブラリ内のモジュールの品質管理

という2つの要素がある．これらはライブラリを構成している要素がモジュールであるために互いに関連をもったものが多いが，ライブラリの場合には，各モジュールをアセンブラによりアセンブルして得られたリロケータブル•オブジェクトモジュールをライブラリマネジャにより結合したものであるために，そのファイル自体をチェックすることはできない．そこで，ライブラリの品質管理とは，おもにそのドキュメンテーションのチェックと，ライブラリマネジャを使用したライブラリ内の登録モジュール名のチェックおよび特別なテストプログラム(テストスタブ)による機能チェックとなる．それに対して，モジュールの品質管理の場合には，そのソースプログラムが存在するので，これを含めたプログラムチェックと考えてもよい．すなわち，これら2つの品質管理は目的は同じでも，その方法には大きな相違がある．

◎　ライブラリ自体の品質管理

[ライブラリの独立性]

ライブラリ内のモジュールの機能は，すべて，そのライブラリ内で満足されているか.

⟶　他のライブラリ内のモジュールを使用してはいけない.

[システムとの関係]

そのライブラリを使用できるシステムの条件が特定のものになっていないか.

⟶　メモリの容量，外部記憶装置等に特別の条件があってはいけない.

[他のライブラリとの関係]

共通に使用するメモリエリア，システムコール等の名称は統一されているか.

[機能の統一性]

ライブラリ内の機能は，統一性があるか.

⟶　BDOS に関するもの，演算に関するものというように，その内容に統一性をもたせることが必要.

[ライブラリのドキュメンテーション]

そのライブラリに関するドキュメンテーションは完全であるか.

⟶　修正を行なった場合の履歴などが明確に記録されているか.

◎ モジュールの品質管理

[信頼性に関するチェック]

・ループや分岐処理は，モジュール内で収束しているか.
・そのライブラリ内に使用するすべてのモジュールがあるか.
・入出力条件は明確化されているか.
・エラー発生時の処理はなされているか.
・モジュール単体でのテストは完全か.
・ラベル名の重複はないか.
・コメントやフォーマットのスタイルは統一されているか.
・ライブラリ内での共通の名称は統一されているか.

[拡張性に関するチェック]

・モジュール名，ラベル名には拡張性が考慮されているか.
・モジュールは，1つの機能からなっているか.
・モジュール内の論理的な流れは，つねに上から下に流れるようになっているか.
・すべてのモジュールの入口と出口は1つになっているか.
・モジュールのサイズに制限を設けてあるか.
・モジュール間のパラメータの受渡しは統一がとれているか.

[保守性に関するチェック]

・変数名等のラベルには，その特徴を示す名称が付けられているか.
・異常終了(エラー等による)の条件，およびその場合の処理方法が説明されているか.
・変数のクロスリファレンス・リストがあるか.
・サブルーチンの呼び出し関係を示したマップは用意されているか.
・そのモジュールで使用しているその他のモジュール等が示されているか.
・コメント文のフォーマットは統一されているか.
・変更が予想される部分に関して，その変更により影響を受ける部分と，その影響について，明確に示されているか.
・バージョンの変更により変化した部分が明確に示されているか.

[移植性に関するチェック]

・特殊なハードウェアに関する機能を使用していないか.
・変数名などに特殊な名前を使用していないか.
・ROM 内ルーチンなどの特殊な条件下での処理を行なっていないか.

◇　品質管理のノウハウは？

チャチャ丸　これは今流行のQCですね．ここにまとめられていることは，単にライブラリの品質管理というよりも，プログラム開発の品質管理といってもよいのではないですか？

Mr.じゅげむ　もちろん，広義に解釈すればプログラム開発のための品質管理ということがいえるね．しかし，ライブラリといっても使用する言語でその管理の方法にはかなりの相違があるんだよ．ここで君に解説しているのは，マクロアセンブラを使用して，そのリロケータブルオブジェクト•モジュールをライブラリマネジャによりライブラリ化したものをいっているので，ライブラリ自体の品質管理ということに関しては，かなり独自のものになっているところがあるんだよ．

チャチャ丸　このリロケータブルオブジェクト•モジュール以外でもライブラリ化というのはできるのですか？

Mr.じゅげむ　もちろんできるよ．例えば，アセンブリ言語でもアブソリュートアセンブラしか入手できない場合には，ソースプログラムのライブラリを作成しておいて，これをメインプログラムの中に“INCLUDE命令”によって読込むという方法もあるし，C言語などの場合には，C言語で記述されたソースプログラムをライブラリとして管理するライブラリマネジャが用意されているんだよ．

チャチャ丸　はあ～そんなものですか！　ところで，モジュールの品質管理の解説の中にいくつかよく理解できないところがあるのですが，この解説では，品質管理の項目を4つの要素に分けていますね．すなわち，

1）“信頼性に関するチェック”
2）“拡張性に関するチェック”
3）“保守性に関するチェック”
4）“移植性に関するチェック”

この意味をもう少し詳しく解説してくれませんか？

Mr.じゅげむ　この中で“信頼性に関するチェック”というのは，プログラムを作成する場合の常識的な項目になっているわけだね．次の“拡張性に関するチェック”というのは，作成したモジュールの機能を拡張させたときにそれに対する考慮がなされているかということになる．どうしても納期に追われると拡張性を犠牲にしてしまうことがあるが，これはモジュールの完成度という面からも避けなければならないことだね．この2つのチェックポイントは主にプログラムのアルゴリズムに関することになるわけだが，次の“保守性に関するチェック”というのは，主にプログラムのコメントをも含めたドキュメンテーションに関係するものになる．すなわち，完成したプログラムのメンテナンスを行なう場合に必要な項目をチェックしようということだね．次の移植性に関するチェックというのは，つねに同じハードウェアとソフトウェア環境を使用してい

るのであれば関係ないが，これが変更されるような可能性がある場合には，十分に考慮しなければならない問題だね．

チャチャ丸　なるほどね．なかなか難しい問題があるのですね．

Mr. じゅげむ　だから，先のレクチャーの部分もある程度の目安と考えたほうがよいだろうね．

◇

品質管理というのは，プログラムの開発から，メンテナンスまですべてを手がけていないと，なかなか，そのカンどころというものがわからないし，単に管理のために，規則をきびしくすれば，こんどは生産性が悪くなってしまう．結論的にいえば，「こうすれば完璧である」という方法はないが，チェック・ポイントをレクチャーのように分けて考えることによってかなり問題点が明確になってくるということだろうね．

これについてさらに勉強をするのであれば，参考文献の6を一読してみることだネ！

第 2 章

マクロ・アセンブラの使用法のテクニック

[この章のポイント]

MACRO–80 を使用してマクロ・ライブラリを作成する際に必要となる基本的知識とマクロ定義のテクニック，およびライブラリ作成の注意点に関して次の手順で解説する．

(MACRO–80 の文法に関しては第 3 章を参照のこと)

◎ マクロ・ライブラリの有効性

◎ マクロ定義の方法

◎ マクロ定義のアセンブルのされ方

◎ パラメータの受渡しの方法

◎ マクロ定義に関するテクニック

◎ マクロ・ライブラリ作成の注意点

2.1 マクロ・ライブラリの有効性

プログラムをライブラリ化して開発することの必要性と有効性に関しては，第1章で解説した．このライブラリ化したプログラム，すなわちモジュールをより効率よく活用するのがこのマクロ・ライブラリということになる．一般的には，マクロ・ライブラリというとディジタル・リサーチ社のMACのようにリロケータブルでないアセンブラでも，マクロ定義の可能なアセンブラであればこのマクロ定義の部分をライブラリ・ファイルとしてこれを“マクロ・ライブラリ”と呼ぶことが多い（このMACを使用したマクロ・アセンブラの使用法に関しては参考文献5が特筆に値する……ぜひ一読を薦める）．

本書でも一部分はこのようなマクロ定義を集めただけのマクロ定義ライブラリを使用するが，本書の主たる目的は，これらのマクロ定義部と，そのマクロ定義部から呼び出されるリロケータブル・モジュールの管理を含めたマクロ・ライブラリの作成法ということになる．すなわち，

| マクロ定義ライブラリ | ＋ | リロケータブル・モジュール |

↓

| 本書でのマクロ・ライブラリ |

という形が本書の目的である“マクロ・ライブラリ”なのである．この方法のメリットはどこにあるのであろうか？

まず，リロケータブル・アセンブラを使用してプログラムをリロケータブル・オブジェクトモジュールにすることにより，すでにチェック済のプログラムを再度アセンブルするという作業を省略することができる．これは，ラベルの二重定義などという初歩的なミス（共通参照記号や外部参照記号などにこのミスを犯すことはほとんどないが，ローカルなラベルにおいてはしばしば犯してしまうミスである）を防げるとともに，アセンブルを行なうよりも，オブジェクト・モジュールをリンクする時間のほうがはるかに短くてすむ．これは短いプログラムのときにはアセンブル時間というのはあまり気にならないが，シンボルの数が多いプログラムをアセンブルすると思いのほか時間がかかるし，アセンブル・リストが占めるファイルの容量というのもバカにならない．さらに，そのリストをプリンタに出力するとなると不用なリスト（すでに，チェック済のモジュール部分）まで出力されてしまい時間と用紙の無駄になるばかりである．しかも，チェック済のモジュールを再度アセンブルするというのは，市民権を得たものを再度白紙にもどして再チェックするようなもので，作業効率の点からもあまり好ましいことではない．

次にこのライブラリ・モジュールの活用ということを考えてみると，これらのモジュールは一

般的に“サブルーチン”の形をしたものになっているはずである．ということは，これらのモジュールを使用するときには，次のような形式をとることが考えられる．

```
LD   A, **
LD   HL, **        そのモジュールに必要なパラメータの設定
  ~
CALL  ****         モジュールの呼び出し
```

このような形式で記述されているソース・プログラムでは，よほどコメントが丁寧に記述されていないと，どこまでがメイン・プログラムで，どこからがパラメータの設定部分なのかきわめて不明確なものになってしまう．それに対してこの部分を，

```
モジュール名          パラメータ・リスト
----------------------------------------
例：  STRING          D$  ←── パラメータ
        ↑
        └──── モジュール名（実際にはマクロ名）
```

というような記述ができるならば，このプログラムはきわめて見通しのよいものとなる．これは，モジュールを利用する際のソース・プログラム上でのパラメータの設定部分にマクロ呼び出しを利用することによって可能となるのである．このような方法を使用すれば，あたかも高水準言語のプログラムと見まちがえるようなプログラムをアセンブリ言語で記述することも可能となるのである（サンプル・プログラム参照のこと）．

もちろんこのような記述を行なうためには，マクロ定義部でそれに対応する定義をしておかなければならない．しかし，これは一度定義したものをマクロ定義ライブラリ・ファイルの形にしておき必要なときに“INCLUDE 擬似命令”を使用すれば，再度マクロ定義文を記述する必要がなくなるのである．このように，マクロ定義内で外部のリロケータブル・モジュールを使用するには“2.5 マクロ定義に関するテクニック”にまとめてあるようなテクニックが必要になることはいうまでもない．しかし，このテクニックというのは特別なものではなく，

> マクロ定義部では単に外部のリロケータブル・モジュールで必要になるパラメータを設定するだけで，実際の処理はその外部モジュールが行なうようにする．

ということだけである．これにより，マクロ本体が小さくなるために，マクロを使用したとき

にしばしば問題となる，「ソース・プログラムの大きさに対してオブジェクト・プログラムが思いのほか大きくなる」という現象を最小限にくい止めることができるというメリットさえ生まれてくるのである．またこれは，マクロ定義を使用したプログラム全体にいえることであるが，本来その CPU には備わっていない命令を付加することもできるし，マクロ定義を効率よく利用すればメイン・プログラムは処理の流れを表わすだけのものというような記述も可能になる．このような形でメイン・プログラムを記述するようにすれば，オペレーティング・システムのシステム・コールの詳細を知らない者やアセンブリ言語でのプログラミングに不慣れの者でも，そのプログラムの内容を把握することが可能であるし，また，そのプログラムの開発に参加することも可能となる．これは，マクロ定義部が実際の処理部分とのインターフェースの役割を果たすことから生ずるメリットということができる．このことがデバッグを含めて，プログラムの開発効率の向上に果たす役割は容易に理解できるであろう．

また，このリロケータブル・マクロ・ライブラリによるプログラム開発の手法は，その記述性のよさから高水準言語によるプログラム開発と比較することができるが，あくまでもこのリロケータブル・マクロ・ライブラリは，アセンブリ言語によるプログラム開発の効率化を図ったものであり，記述性ばかりを追及したような冗長性の高いマクロ定義は好ましくない．そのような場合には，最初から高水準言語を使用すればよいのである．

これらのリロケータブル・マクロ・ライブラリによるプログラム開発のメリットをまとめると，次のようになる．

- リロケータブル・オブジェクト・モジュールを効率よく利用できる．
- CPU のもつ命令以外の命令をつくることができる．
- ソース・プログラムの記述性がよくなる．
- プログラムのメンテナンスが容易になる．

2.2　マクロ定義の方法

マクロ・アセンブラでマクロ命令(マクロ呼び出し)を使用する場合には，前もってそのマクロ命令に対する“マクロ定義”がなされていなければならない．マクロ定義がされているとアセンブラは，そのマクロ名(マクロ定義をしたときに付けた名前)がオペレーション・フィールドに現われるたびに，その定義部の内容に従ってソース・プログラム上に命令を展開していく．

マクロ定義の基本形は，次のようになる．

シンボル・フィールド	オペレーション・フィールド	式フィールド
マクロ名	MACRO	ダミーパラメータ
↑		
マクロ呼び出しに使用する名称	マクロの定義内容	← この部分が展開される（マクロ定義部）
	ENDM	

このマクロ定義は，式フィールドの指定はオプションで，ここにダミーパラメータ(オペランド)をもつかもたないかで2つのパターンに分けられる．これは，Z80の基本命令にも“LD　A，B”のようにオペランドをもつものと，“NOP”のようにオペランドをもたないものがあるのと同じである．このマクロ定義部にはCPUのもつ基本命令や，一般の擬似命令だけでなく，条件付き擬似命令をはじめ，ブロック擬似命令などを記述できるのはもちろんのこと，マクロ定義さえも記述することができる(マクロの入れ子)．これらの複雑なマクロ定義に関しては，“2.5マクロ定義に関するテクニック”に解説してあるので，ここでは最も基本的なマクロ定義の2つのパターンを説明することにする．

2.2.1　パラメータをもたないマクロ定義

この形のマクロ定義は最も簡単なもので，定義部に記述されたものがそのままソース・プログラム上に展開される．

すなわち，このパラメータをもたないマクロ定義というのは，

マクロ定義部で指定した文字列の置き換え．

と考えてもよい．これを実例で考えてみよう．例えば，Z80のすべてのレジスタの退避と復帰を考えてみよう．Z80には，レジスタ・ペアでの退避(PUSH　**)と復帰(POP　**)はあるが，すべてのレジスタに対してこれを行なう命令は用意されていない．そこでこれを“APUSH・A

POP”という名前でマクロ定義しようというのである．この目的のためにはオペランド(パラメータ)を必要としない．なぜならば，最初から“すべてのレジスタ”という形でオペランドが固定しているからである．このマクロ定義は図2.1のようになる．このように定義しておくことにより，“APUSH・APOP”という新しい命令ができたのと同じ結果になり，ソース・プログラム上で，これらの命令を記述するたびにマクロ定義で定義した内容と全く同じものが展開されることになる．

```
APUSH    MACRO
         PUSH    AF
         PUSH    BC      定義部
         PUSH    DE
         PUSH    HL
         ENDM
----------------------------------
APOP     MACRO
         POP     HL
         POP     DE      定義部
         POP     BC
         POP     AF
         ENDM
```

図2.1　パラメータをもたないマクロ定義の例

2.2.2　パラメータをもったマクロ定義

この形のマクロ定義は，先のものに比較するとかなり複雑な働きをする．複雑さゆえに高度な利用法があるわけであるが，その複雑さの原因は“ダミーパラメータの動き”ということにある．まずこの“ダミーパラメータ”と“パラメータ”の相違について考えてみよう．

この2つの名前の使い分けを本書では，

マクロ定義のパラメータ	⟶	ダミーパラメータ
マクロ呼び出しのパラメータ	⟶	パラメータ

と分けている．この“ダミー”という言葉は“仮の”と考えてもよい．すなわち，マクロ定義の式フィールドに記述されるパラメータは，マクロ呼び出しで指定されたパラメータを受け取り，定義部のパラメータに渡すための整理番号のようなもので，名称その他には何の意味もない．

例えば，

```
TEST    MACRO   P1, P2    ←── ダミーパラメータ
        LD      A, P1     ←─┤
        LD      HL, P2    ←─┘
        CP      (HL)
        ENDM
```

というようなマクロ定義がしてあり，これを，

```
        TEST    '$', 8000H   ←── パラメータ
```

という，2つのパラメータを指定したマクロ呼び出しを行なうと，先の「P1, P2」というマクロ定義のダミーパラメータは，「'$', 8000H」という実際のパラメータに変換され，

```
        LD      A, '$'       ←─┐
        LD      HL, 8000H    ←─┘ 変換されたパラメータ
        CP      (HL)
```

という形で定義部のダミーパラメータに反映するわけである．もちろん，名称に関係ないといっても，先の「P1, P2」などを「A, HL」というような名称にしてしまうとアセンブラは，定義部に，A とか HL という綴りが出てくるたびにこれを指定されたパラメータに変換してしまうので，

```
        LD      '$','$'          レジスタ名の A, HL も変換されて
        LD      8000H, 8000H     しまう
        CP      (8000H)
```

というような意味不明のものとなってしまうので注意が必要である．このパラメータの設定法には，その他にもいろいろなルールとか，このパラメータにのみ作用する演算子や，一見できそうでできないというようなことがいろいろあり，このパラメータの設定法を完全にマスターすれば，マクロを征服したといっても過言ではない．

これらのことについては，2.4 節と 2.5 節に解説してあるのでそちらにゆずる．

2.2.3　マクロ定義のソース・プログラム上の位置

マクロ定義は基本的には，そのマクロ呼び出しを行なう前であれば，どこで定義してもよい．しかし，特殊な場合を除いてメイン・プログラムに先だってまとめた形でマクロ定義部分として置かれるのが普通である．

すなわち，

マクロ定義部分
メイン・プログラム

という形が一般的なフォーマットということができる．そして，このマクロ定義部分は，1つのファイルとして用意しておくことが多いので，“INCLUDE 擬似命令”を使用して，

INCLUDE　マクロ定義ファイル名
メイン・プログラム

とすることが多い．ただし，これはチェック済の完成したマクロ定義ファイルがある場合で，マクロ定義の開発段階では，前者の形で開発するほうが効率がよい．また，MACRO-80 の INCLUDE 擬似命令は，入れ子ができないので，これらのマクロ定義ファイル内では，この INCLUDE 擬似命令を使用してはならない．

2.3　マクロ定義のアセンブルのされ方

CP/M の ASM のようなマクロ定義をもたないアセンブラの場合には，プログラムをアセンブルするときのオペレーション・フィールドの扱いは，擬似命令と基本命令をチェックしてそれに対応するコードを生成するだけであるが，マクロ定義をもつアセンブラの場合には，このオペレーション・フィールドに“MACRO”という綴りが見つかった場合にはこれをマクロ定義テーブルに格納して，そのマクロ名に対応するマクロ呼び出しがオペレーション・フィールドに現われたときに，そのマクロ定義テーブルの内容を展開しなければならない．

このマクロ定義テーブルへの格納の方法は，ディジタル・リサーチ社の MAC も，このマイクロ・ソフト社の MACRO−80 も大差ないのであるが，擬似命令や基本命令の処理とマクロ定義に対する処理の順番はまるで逆になっている．すなわち，ディジタル・リサーチ社の場合には，まず擬似命令や基本命令をチェックして，次にこのマクロ定義のチェックを行なう．それゆえ，マクロ名としてこれらの擬似命令や基本命令とおなじ綴りのものがあれば，このマクロ名をマクロ名とはみなさずに擬似命令や基本命令として処理してしまう．

それに対して，このマイクロ・ソフト社の MACRO−80 の場合には，まずマクロ定義のチェックから行ない，次に擬似命令や基本命令のチェックを行なう．それゆえ，マクロ名としてこれらの擬似命令や基本命令をマクロ名として使用することもできるのである．極端な例をあげると，マクロ名として“MACRO”という名前さえ使用することができる．これは，どちらの方法が優れているとは一概にはいえないが，

MACRO−80 のほうが自由度が高い．

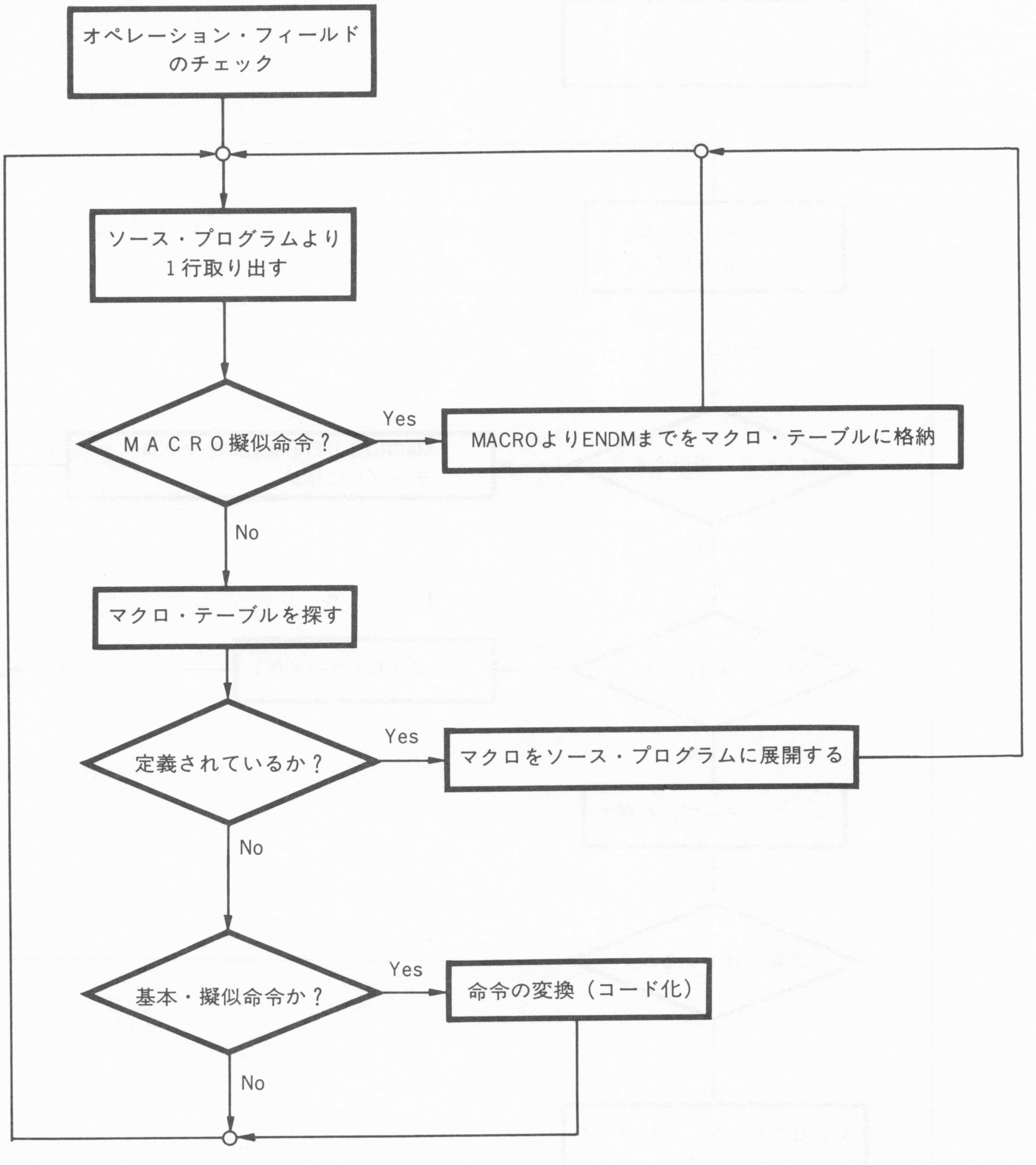
ということはいえるであろう．しかし，この自由度の高さは思わぬ失敗につながることもある．例えば先の例のように“MACRO”というようなマクロ名を使用してしまうと本来の“MACRO 擬似命令”の機能はなくなってしまいとんでもない結果が生ずることも起こりうる．しかしながら逆にディジタル・リサーチ社のような方法であると擬似命令や基本命令に使用されている綴り(名前)は予約語として扱われてしまうので，マクロ定義を利用した“クロス・アセンブラ”をつくるような場合には，これらの名前と同じニーモニックを使用することが不可能となってしまう．このような場合には，MACRO−80 のマクロ名の自由度というのはきわめて重宝なものになる．すなわち，この自由度というものは，両刃のカミソリのようなものでそれを使用する人の力量にかかわってくるといってよいであろう．

また，マクロ定義テーブルへの格納の方法はディジタル・リサーチ社のものも，マイクロ・ソフト社のものもほとんどかわりはないが，これらのものは，定義をされた順にテーブルの中に格納されていく．そして，マクロ呼び出しに出会うとこのテーブルを後方(最後に定義されたも

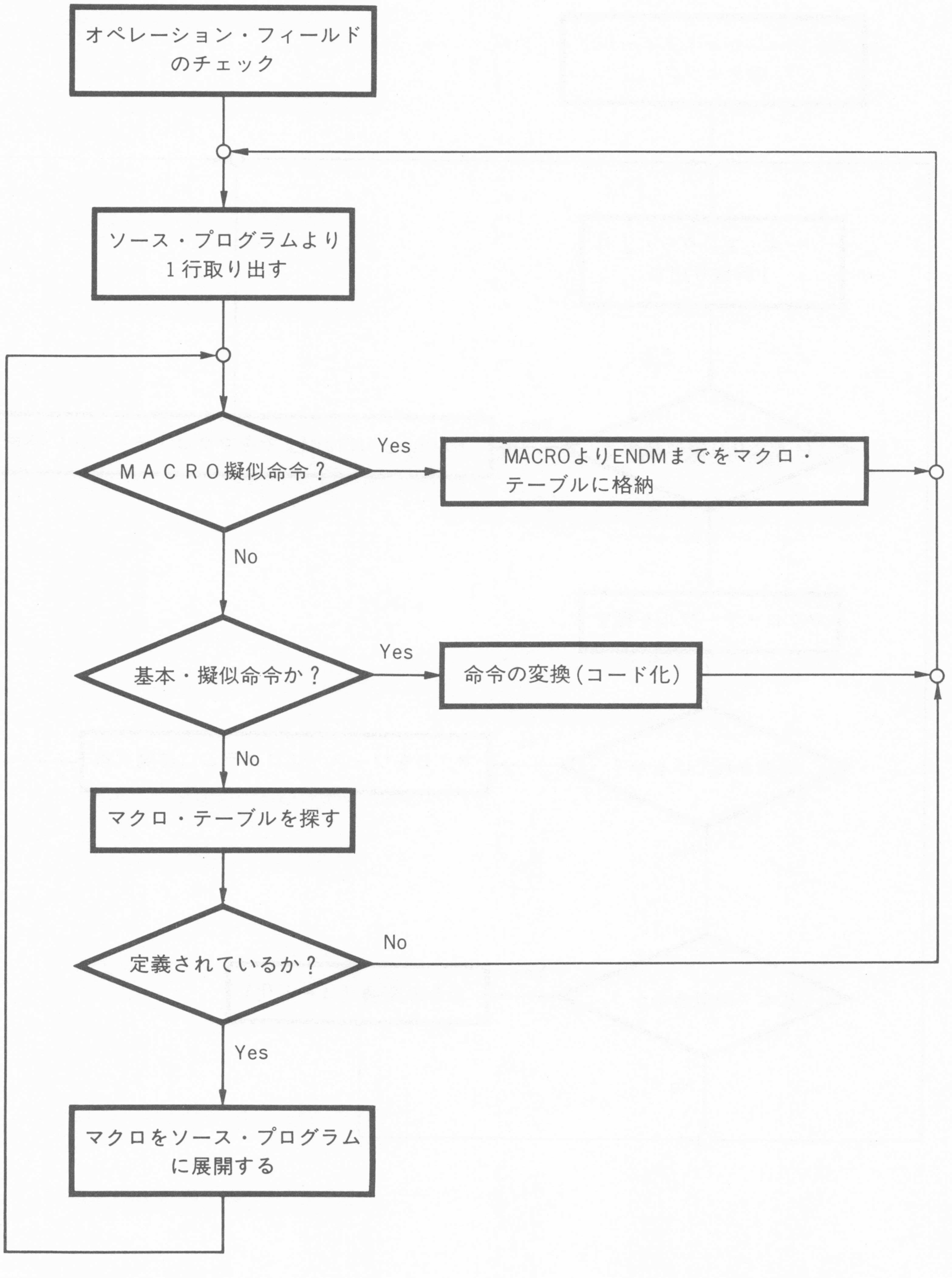
のから)からそのマクロ名が定義されているかチェックしていく．また，マクロ定義の中にさらにマクロ定義がなされていると(マクロ定義の入れ子)，アセンブラはとりあえずその外側のマクロ定義に関する部分をマクロ定義テーブルに格納し，そのマクロが呼び出されたときに，内側のマクロ定義をマクロ定義テーブルに格納する．

それゆえ，この外側のマクロ名と内側のマクロ名を同一の名称にしておくと，最初にこのマクロが呼び出された時点で，この外側のマクロ定義と，内側のマクロ定義が同じマクロ名でマクロ定義テーブル内に存在することになる．その場合，2回目以降にこのマクロ名で，マクロ呼び出しが行なわれると，アセンブラは，マクロ定義テーブルの後方からマクロ名をチェックするのであるから，そのマクロ名に対する処理は内側のマクロ定義に関するものとなる．この性質を利用すると“2.5.3 オープンマクロとクローズドマクロ”で解説してある“クローズドマクロ”という手法を利用することができる．

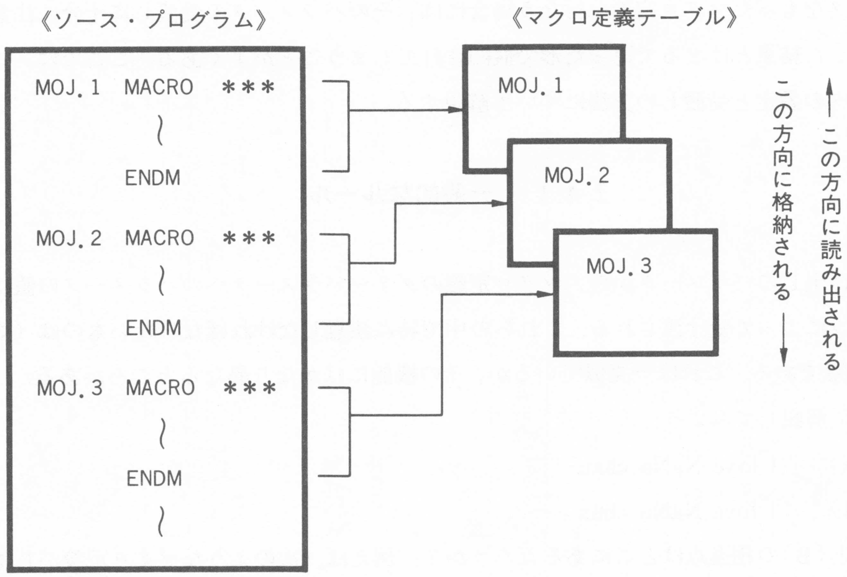
◎ マイクロ・ソフト社のアセンブルのプロセス



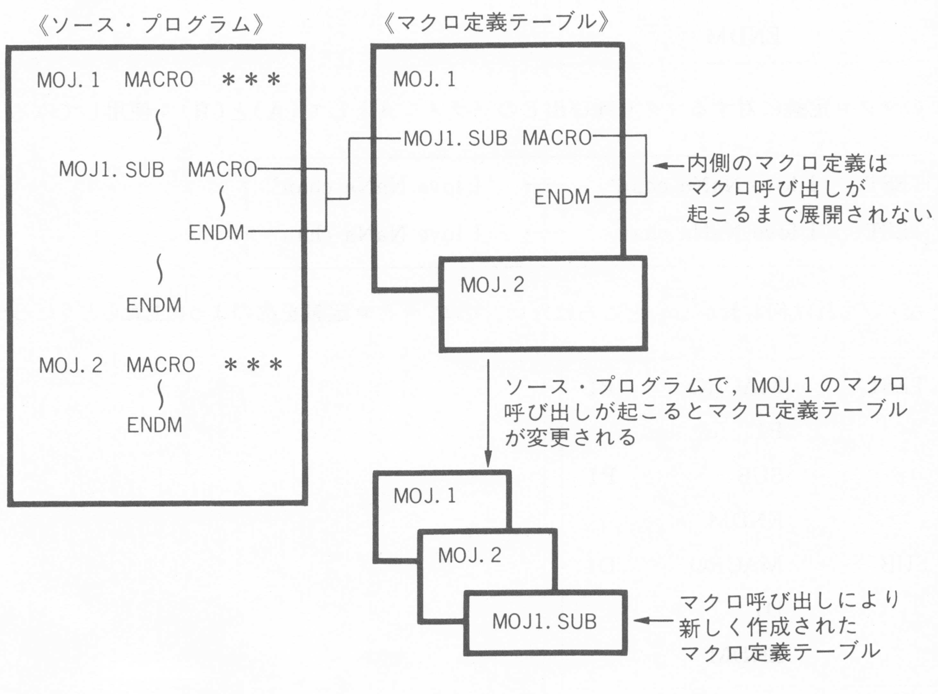
◎　ディジタル・リサーチ社のアセンブルのプロセス



◎　マクロ定義のテーブルへの格納のされ方



◎　マクロ定義の中でのマクロ定義(マクロの入れ子)



2.4 パラメータの受渡しの方法

パラメータをもったマクロ定義を行なう場合には，そのパラメータの受渡しに十分に注意しないと，期待した結果とはまるで違った形で展開されてしまうことがよくある．ここでは，これらのパラメータの設定と受渡しの方法について解説する．

2.4.1 一般的なルール

マクロ呼び出しのパラメータから，マクロ定義のダミーパラメータへのパラメータの値は，**表2.1**のルールによって受け渡される．これらの中で特に注意しなければならないものは（3）と（4）の相違点である．これは一見似ているが，その機能にはかなり異なるところがある．

例をあげて解説してみよう．

（A） 'I love NaNa chan'

（B） <I love NaNa chan>

この(A)と(B)の相違点はどこにあるだろうか？ 例えば，次のようなマクロ定義がしてあるものとする(定義自体はナンセンスでエラーが出る場合もある)．

```
TEST        MACRO       P1
            P1
            ENDM
```

このマクロ定義に対するマクロ呼び出しのパラメータとして(A)と(B)を使用してみる．

```
TEST  'I love NaNa chan'   ──→  'I love NaNa chan'
TEST  <I love NaNa chan>   ──→   I love NaNa chan
```

となり，これは何もおかしいところはない．では，マクロ定義を次のようにしたらどうだろうか？

```
TEST        MACRO       P1
            P1
            SUB         P1
            ENDM
SUB         MACRO       D1
            D1
            ENDM
```

すなわち，TEST というマクロ定義の中で，SUB というマクロ呼び出しを行なっているわけである．これに対して先と同じことを行なう．

```
TEST   'I love NaNa chan'   ←―  マクロ呼び出し
       'I love NaNa chan'   ←―  TEST による展開
       'I love NaNa chan'   ←―  SUB  による展開
.............................................................
TEST  <I love NaNa chan>    ←―  マクロ呼び出し
        I love NaNa chan    ←―  TEST による展開
        I ←―――――――――――――――――  SUB  による展開？
```

このように山形記号で囲んだものは，内側のマクロ展開（SUB）では，愛がなくなってしまう（？）．

これは，最初のマクロ呼び出しから TEST というマクロ定義へのパラメータには，山形記号が付いているから「I love NaNa chan」までが 1 つのパラメータとしてみなされるが，TEST の中で SUB に渡されるときには，この山形記号は取り除かれている．それゆえ，SUB に渡されたパラメータは，「I」，「love」，「NaNa」，「chan」という 4 つのパラメータになってしまっているのである．

ところが，SUB のダミーパラメータは 1 つしか記述されていないから， 表 2.1 の（6）のルールから「愛がなくなる」ということになってしまったわけである．この場合，TEST のマクロ定義を，

```
TEST        MACRO       P1
            P1
            SUB         <<P1>> ←  SUB でのマクロ展開のために山形記号を付け
            ENDM                  る
```

としておけば問題はない（愛は大切に山形記号で囲むべし！！！）．

上記のような場合にはすぐに気づくであろうが，これは条件付きマクロや，少々複雑なプログラムになると意外な盲点となるので注意してほしい．

少し複雑なパラメータを使用するときには，必ずこの山形記号で囲んでおくという習慣をつけておくとよいかもしれない．

表 2.1 パラメータ受渡しの一般的ルール

(1)	複数のパラメータがある場合には，それぞれのパラメータは，カンマ(,)，空白(20H)，セミコロン(;)，エクスクラメーション(!)，あるいはリターン・コード(CR)によって区切られる． 例： I am a boy というパラメータは， I, am, a, boy という4つのパラメータになる．
(2)	パラメータ中の意味のない空白や，タブ記号をはじめとするコントロール・コードは，すべて取り除かれる．
(3)	(')または(")で囲まれたパラメータは，その中に(1)の区切り記号を含んでいても，その記号も文字と見なして，(')または(")までを含めて1つのパラメータとして扱われる． 例： 'I am a boy' というパラメータは，そのまま 'I am a boy' という1つのパラメータになる．
(4)	(＜)と(＞)で囲まれたパラメータは，(3)の場合と同じようになるが，両端の(＜)と(＞)記号は取り除かれる．この山形記号が複数組付いている場合は，1回の受渡しで1組ずつ取り除かれる． 例： ＜I am a boy＞ というパラメータは，両端の記号がとれて I am a boy という1つのパラメータになる．
(5)	マクロ呼び出し側のパラメータの数が，マクロ定義部分のダミーパラメータの数より少ない場合は，NUL が渡される．
(6)	マクロ呼び出し側のパラメータの数が，マクロ定義部分のダミーパラメータの数より多い場合は，無視される．

2.4.2 マクロ演算子の使用法

パラメータの受渡しに関するマクロ演算子としては，次の3つのものがある．

&・・・文字列やシンボルの連結
!・・・1文字を対象として山形記号と同じ
%・・・パラメータの数値を表わす

これらの演算子の中で，3番目のものは，前2者とは使用法が異なり，マクロ呼び出しのパラメータのみに使用できる演算子で，前2者は，マクロ定義部分のダミーパラメータでも，マクロ呼び出しのパラメータでも使用できる演算子ということである．

著者が最もよく使用するのは“&”演算子で，この演算子によりマクロ定義内で用意した文字列と，マクロ呼び出しのパラメータとを結合して，新しい文字列をつくることも可能となる．このような表現では漠然としているので，具体例をあげて解説してみよう．

(A)　& 演算子の使用例

この演算子は，先にも述べたように，マクロ定義内の文字列とマクロ呼び出しのパラメータを結合する機能があるわけであるが，これをもう少しかみくだいて表現すると，

この & 演算子から次の区切り記号までは，ダミーパラメータですよ！

という宣言であると考えてもよい．
つまり，

```
DSET1     MACRO     NO
          LD        A, DATA.NO      ←┐  注意！
          ENDM                       │
---------------------------------------│--------
DSET2     MACRO     NO                 │
          LD        A, DATA.&NO     ←┘
          ENDM
```

というようなマクロ定義がされていたとする．このマクロ定義を同じパラメータを与えて呼び出してみよう．

すなわち，'NO' と '&NO' の相違をチェックしようというわけである．そのために，次のようなソース・プログラムをアセンブルしてみる(もちろん，上記のマクロ定義はしてあるものとする)．

```
DATA.1    EQU       0AH
          DSET1     1      ←┐  パラメータを1としたマクロ呼び出し
          DSET2     1      ←┘
```

このアセンブル結果は，次のようになる．

```
                       DSET1   1   ← NO に対するパラメータ
U  0000'  3E 00    +   LD      A, DATA.NO   ← DSET1 に対する展開
                       DSET2   1   ← NO に対するパラメータ
   0002'  3E 0A    +   LD      A, DATA.&1   ← DSET2 に対する展開
```

DSET1 に対する展開部は，マクロ呼び出しで与えたパラメータが全く反映されていない．す

なわち'NO'というパラメータは，単に'DATA.NO'という文字列の一部としか認識されていないわけである．それに対して，DSET2の展開部には，マクロ呼び出しで与えたパラメータが反映され'NO'が'1'と変換されて「LD　A,DATA.1」となり，「DATA.1」の値「0A」が代入され「LD　A,0AH」のコード「3E　0A」というオブジェクト・コードが展開されている．すなわち，'NO'をパラメータとして認識したわけである．

また，この演算子は文字列の中間部分をパラメータとして認識させることもできる．例えば，

```
TEST    MACRO    P1
        DC       'I love &P1& forever.'
        ENDM
```

とマクロ定義しておき，パラメータとして「NaNa」を与えれば，

```
TEST  NaNa    →    DC  'I love NaNa forever.'
```

という展開が得られる．

この&演算子で注意することは，クォート記号(')で囲んだところ以外は，先の例のように&マークがリスト上には残っているが，これは単なる目印でオブジェクト・コード上には現われないので心配する必要はない．

(B)　！演算子の使用例

この演算子の機能は山形記号と非常によく似ているが，この！演算子の場合には，それに続く**1文字にのみ有効**となる．それゆえ，この演算子を使用することによって，パラメータの区切り記号となっているカンマやセミコロンを文字として使用することができる．しかし，著者は，このようなときにもプログラムの見通しのよさを考えて，山形記号を使用することにしている．

この演算子の機能を別の言葉で表わすと，次のようになる．

パラメータとして使用できないコード(1バイト)を文字化する.

この演算子の使用例を，山形記号を使用した場合と比較して次にあげておく．

≪！演算子を使用した場合≫		≪山形記号を使用した場合≫	
IRP	X,<!,,!　,!;,!!>	IRP	X,≪,>,<　>,<;>,<!≫
DB	'&X&'	DB	'&X&'

```
ENDM                            ENDM
DB        ','                   DB        ','
DB        ' '                   DB        ' '
DB        ';'                   DB        ';'
DB        '!'                   DB        '!'
```

（C）％演算子の使用例

この演算子は，これに続くパラメータを数値として取り扱いたいときに，マクロ呼び出しのパラメータに付けて使用する．すなわち，

マクロ呼び出しのパラメータの数値化

ということになる．しかし，これはシンボルを使用しているときにはあたりまえのことで，この演算子の機能はきわめて不鮮明である．例えば，次の例の場合には，このマクロ呼び出しを％演算子を付けて「DATA　%D1」としても，付けずに「DATA　D1」としてもリスト上で異なって見えるだけで，オブジェクト•コードは全く同じことになる．すなわち，この例では，この％演算子は全く意味がない．

```
D1          ASET        5
DATA        MACRO       P1
            DB          P1
            ENDM
```

では，どのようなときにこの％演算子が効力をもってくるのであろうか？　次の例を考えてみよう．

```
D1          ASET        5
DATA        MACRO       P1
MSG&P1      DC          'NaNa chan'
            ENDM
```

この場合に，先と同じマクロ呼び出しを行なうとどうなるであろうか？　これは容易に想像がつくであろう．

このアセンブル例は，次のようになる．

```
                DATA        %D1
+   MSG&5       DC          'NaNa chan'  ←  DATA  %D1  に対する展開
                DATA        D1
+   MSG&D1      DC          'NaNa chan'  ←  DATA  D1   に対する展開
```

このように全く異なる形で展開される．であるから，この演算子は，

> マクロ定義内で，呼び出しのパラメータの値(数値)をそのまま使用したいときに，この%演算子を呼び出しパラメータに使用する．

ということになる．これは，マクロ展開により新しいシンボルを作成するときなどの特殊な使用法になる．ちなみに，著者はこの演算子をあまり使用したことがない．

2.5　マクロ定義に関するテクニック

マクロ•ライブラリのもとになるマクロ定義は，単に1つ1つ文法どおりに定義をしていってもできないことはないし，また，最初のうちは新しい言語でもつくっているような気がして大変楽しいものである．しかし，このマクロ定義を無秩序に行なっていくと思いのほかアセンブルに時間がかかったり，アセンブルのためのメモリが足りなくなってしまったり，オブジェクト•プログラムが大きくなってしまいプログラムのスループットが悪くなってしまうことがたびたびある．そこで，ここでは，効率よくマクロを定義するためのいくつかの基礎知識とテクニックを解説することにする(文法の詳細は第3章を参照のこと)．

2.5.1　ブロック(マクロ)擬似命令の使用法

マニュアル上のマクロ関係の擬似命令としては，これまで解説してきた“マクロ定義擬似命令”と，この“ブロック擬似命令”がある．この両者の相違は，前者がその機能を自由に定義することができるのに対して，後者は，その機能自体はほぼ固定されていて，繰り返し処理に使用するような形になっている．

これらのブロック擬似命令としては，次の3種類のものがある．

REPT	(Repeat)
IRPC	(Indefinite Repeat Character)
IRP	(Indefinite Repeat)

これらはいずれも繰り返し処理を行なうもので，“ブロック擬似命令”というよりも，“リピート擬似命令”と呼んだほうがよいものである．これらは，マクロ関係の擬似命令といっても，マクロ定義擬似命令のように“呼び出し”という観念はない．つまり，「定義する」のではなくて，「単に用意されている機能を，それが必要な時点で利用する」だけなのである．であるから，本書ではこの命令をマクロ定義命令とは分けて解説している．

さて，これらの命令の使用法であるが，単純にソース•プログラム上に記述する方法もある．このような形でこの命令を使用したときには，単にソース•プログラムの記述上の手間を省く効果しかない．しかし，マクロ定義内でこれらのブロック擬似命令を使用することによって，これらの命令の機能が何倍にも拡張する．ここでは，これらの方法を考えてみることにする．

(A)　REPT の使用例

例えば，デバッグ用に10個の NOP 命令を記述することを考えてみよう．これは単純に，

```
        NOP  ─┐
        NOP   │
        ~     │ 10個の NOP 命令
        NOP  ─┘
```

とこれまでは記述していた．これは，エディタのコピー機能を利用して記述するのが一般的であろうが(もちろん，1つ1つ記述してもよいが……)，このようなときにこのブロック擬似命令を使用するともっとスマートに記述できる．それは，次のように行なう．

```
        REPT    10  ←  以下 ENDM までの操作を 10 回行なう
        NOP
        ENDM  ←  必ず必要である
```

これで10個の NOP 命令がオブジェクト・プログラムに展開される．これは最も単純な例であるが，この単純な例も少し工夫することによってもう少し利用価値が高まる．それは，この**ブロック擬似命令をマクロ定義内で使用する方法**である．先の例では，展開する必要のある場所でいちいち3ステップの命令を記述しなければならない．これは，プログラムの見通しという点からはあまりよくない．「NOP を10個」という単純な要求なのだから記述上の表現もなるべく簡単なほうがよい．そこで，これを次のようにマクロ定義しておく．

```
GNOP    MACRO   P1    ;;GNOP:Generate NOP
        REPT    P1
        NOP
        ENDM    ←REPT に対する ENDM
        ENDM    ←MACRO に対する ENDM
```

このように定義しておくことによって，複数の連続した NOP が必要になったところで上記のマクロを呼び出せばよい．すなわち，「GNOP 10」という記述で10個の NOP 命令を記述するのと同じ結果が得られるのであるから，プログラムの見通しがよくなることは当然である．このようにブロック擬似命令をマクロ定義内で使用することによりマクロ定義の有効利用にもなるし，また，ブロック擬似命令も利用しやすい形に生まれ変わることができるわけである．

(B) IRP の使用例

先の例では，展開の回数だけを指定する最も簡単なブロック擬似命令である REPT を使用し

たが，次にIRPブロック擬似命令の使用例を考えてみよう．この命令の特長は，パラメータ列から1パラメータずつ取り出しダミーに代入するものであるから，それを利用して次のようなマクロ定義を行なってみる．

```
APUSH     MACRO
          IRP       X,<AF,BC,DE,HL>
          PUSH      X
          ENDM
          ENDM
```

これは，どのように展開されるだろう．ダミーパラメータXが，AF, BC, DE, HL に置き換えられていくわけだから，すべてのレジスタに対するPUSH命令ということになる．すなわち，“2.2.1 パラメータをもたないマクロ定義”で例として取り上げたものと同じことになるのである．さらにこれを拡張して，

```
@PUSH     MACRO     P1
          IRP       X,<P1>
          PUSH      X
          ENDM
          ENDM
```

としたらどうであろう？　例えば，このマクロ定義を“@PUSH ＜BC, DE＞（パラメータを囲む山形記号に注意）”というパラメータで呼び出すと，どのように展開されるであろう．

1. マクロ呼び出しのパラメータ<BC, DE>が，MACROのP1に渡される．
2. この時点で両端の山形記号がはずされるから，P1はBC, DEとなる．
3. それがIRPのP1に渡されるから，このIRP擬似命令は，次のようになる．
```
          IRP       X,<BC,DE>
          PUSH      X
          ENDM
```
4. そして，これが以下のように展開される．
```
          PUSH      BC
          PUSH      DE
```

このマクロ定義の例などは十分に実用になる．これらのことをブロック擬似命令を使用せずに行なうとしたら，かなり面倒なものになるであろう．

(C)　IRPC の使用例

この命令は，パラメータに指定した文字列から，1文字ずつ取り出してダミーに代入するものであるから，これを利用して次のようなマクロ定義をしてみる.

```
CLEAR      MACRO      P1
           IRPC       REG, P1
           LD         REG, 00
           ENDM
           ENDM
```

この定義に対して“CLEAR　BC”というマクロ呼び出しを行なうと，次のようなプロセスでマクロ展開が行なわれる.

1. マクロ呼び出しのパラメータ BC が，MACRO の P1 に渡される.
2. P1 が BC となるから，この IRPC 擬似命令は，次のようになる.

```
           IRPC       REG, BC
           LD         REG, 00
           ENDM
```

3. そして，このパラメータから1文字ずつ REG に渡され次のように展開する.

```
           LD         B, 00
           LD         C, 00
```

これは，次のように IRP 擬似命令を使用しても実現できる.

```
CLEAR      MACRO      P1
           IRP        REG, <P1>
           LD         REG, 00
           ENDM
           ENDM
```

と定義しておき，これを呼び出すときのパラメータの設定は，次のようにする.

```
CLEAR      <B,C>        ←  パラメータの山形記号は必ず付ける
```

どちらがよいかは何ともいえないが(呼び出しパラメータの見通しのよさでは，IRP を使用したほうがよいと思うが)，IRPC 擬似命令を使用したものは，IRP 擬似命令によりほとんどのものが実現できるが，その逆は不可能なものが多い．IRPC 擬似命令の特長は，パラメータの文字列を 1 文字ずつに分解するところにあるのだから，それを利用するような場合に使用するのがポイントといえる．

(D)　ブロック擬似命令の活用のポイント

これまで解説したように，これらの擬似命令は単独で使用することもできるが，マクロ定義の中で使用することによって，さらに機能を高めることができる．またこれらの擬似命令は，次の条件付き擬似命令との併用によりこの命令をコントロールでき(条件により展開を制御できる)，さらに付加価値の高いものとなる．

これらの活用のポイントと注意点をまとめると，次のようになる．

1. 各命令の機能を認識する．
   A) REPT は，パラメータをもたない単純な繰り返しに使用する．
   B) IRP は，複数の文字列をパラメータとしてもつものの繰り返しに使用する．
   C) IRPC は，文字列をパラメータとしてもち，その文字列の 1 文字ずつが操作の対象となるときに使用する．
2. マクロ定義内で使用することにより，これらの機能をマクロ呼び出しという形で実現させせることができる．……(マクロ定義内での利用法)
3. マクロ定義内で使用するときにも，この擬似命令の ENDM 文が必要になるので忘れないこと．……(ブロック擬似命令の終端は ENDM となる)
4. 文字列とパラメータの結合などに使用するマクロ演算子は，この擬似命令の中でも使用できる．

```
例：  IRP    REG,<A,B,C>          DB   'A'
      DB     '&REG&'        →     DB   'B'
      ENDM                        DB   'C'
```

5. 条件付き擬似命令との併用により，さらに機能を高めることができる．

```
例：  STACK  MACRO    SW,DATA
             IFIDN    <SW>,<ON>
               IRP    REG,<DATA>  ┐
               PUSH   REG         │ SWがONの場合
               ENDM               ┘
```

```
        ELSE
          IRP     REG,<DATA>  ┐
          POP     REG         │ SWがON以外の場合
          ENDM                ┘
        ENDIF
        ENDM
         ≪使用例≫
        STACK     ON,<BC,DE>
+       PUSH      BC
+       PUSH      DE
        STACK     OFF,<DE,BC>
+       POP       DE
+       POP       BC
```

2.5.2 条件付き擬似命令の使用法

この条件付き擬似命令に関しては，ディジタル・リサーチ社のMACやRMACのほうが，はるかに使いやすくなっている．それは，このディジタル・リサーチ社のアセンブラの条件判断を行なう命令語は高水準言語でおなじみの"IF"で，IFの後に条件を記述して，その結果により処理を行なうというFORTRANやBASICでおなじみの形態になっている．それに対して，このMACRO−80の条件付き擬似命令というのは，**判断を行なう条件により命令語自体が異なっており**，どの命令語がどのような条件判断を行なうのか覚えていないと使用できない．つまり，このMACRO−80の場合には，命令語により条件判断の機能が決められているので，その条件判断の範囲には限界があるということになる．

言葉を換えて表現すると，

> MACRO−80の条件付き擬似命令には，条件文(式)というのはなく，命令語固有の条件判断のためのパラメータを指定できるだけの機能しかない．

といってもよいであろう．これは大変残念なことで，これによりプログラムの見通しがきわめて悪いものになってしまう．というのは，これらの条件判断用の命令語は，真の判定用と偽の判定用のものがあるのだが，これらの命令語の綴りがきわめて不明確で，著者もこれを逆に用いて大失敗をしたことがある．

このMACRO−80の条件付き擬似命令のフォーマットをまとめると，次のようになる．

```
IF****        [パラメータ]
 ↑               └─ 条件判断のためのパラメータ(条件文ではない！)
 └────────────── 判定条件を表わす命令語
    ┌─────────── 条件が満たされるとこの部分がアセンブルされる
  ～ │
    └─
ENDIF
```

このように，MACRO－80の条件付き擬似命令というのは，パラメータ部分に，

```
IF      A>B
```

というような条件式を書けないために，一般の条件判断には使いにくいところがあるが，一方では，ディジタル・リサーチ社の条件付き擬似命令には用意されていない次のような特長もある．

> アセンブルの過程を調べたり，シンボルの定義状況をチェックできる．

これは具体的には，アセンブルの過程をチェックする“IF1・IF2”，シンボルの定義状況をチェックする“IFDEF・IFNDEF”というようなものになるが，これらの機能をもつ擬似命令はディジタル・リサーチ社のものにはない．それゆえ，これらの特長を十分に理解して，この擬似命令を使用するのがポイントとなる．

このMACRO－80の条件付き擬似命令の機能をまとめると，次のようになる．

> パラメータが真(ゼロ以外)か偽(ゼロ)かを調べる・・・・・・・・・・IF, IFE
> アセンブルの過程を調べる・・・・・・・・・・・・・・・・・・・・・・IF1, IF2
> パラメータとして与えたシンボルが定義されているかどうかを調べる・IFDEF, IFNDEF
> パラメータがNULまたは空白かどうかを調べる・・・・・・・・・・IFB, IFNB
> パラメータとして与えた2つの文字列が等しいかどうかを調べる・・・IFIDN, IFDIF

これらの擬似命令は単独で使用することもできるが，ほとんどはマクロ定義擬似命令およびブロック擬似命令の定義部分で使用することが多い．次にこれらの条件付き擬似命令を機能別に分けた使用例を解説する．

(A)　パラメータの真と偽を調べる例(IF，IFE)

この機能はASMでもおなじみのもので，ソース・プログラム上で単独で使用することも多い

が，ここでは，マクロ定義内で使用した例を示すことにしよう．

この擬似命令をマクロ定義内で使用することによって，PASCALなどに用意されている「CASE文」に相当する処理を行なうことができる．ここでは，BASICの，

```
ON　スイッチ　GOTO　処理1，処理2，・・・
```

に対応するような形で定義してみよう．この処理を考える場合，スイッチの状態を何通り設定するかによって，処理番号を表わすパラメータの処理が変わってくるが，ここでは，簡単に3個という固定の数にした．

```
CASE    MACRO    SW, L1, L2, L3
        IFE      SW-1   ┐
        JP       L1     │ if  SW=1  go to  L1
        ENDIF           ┘
        IFE      SW-2   ┐
        JP       L2     │ if  SW=2  go to  L2
        ENDIF           ┘
        IFE      SW-3   ┐
        JP       L3     │ if  SW=3  go to  L3
        ENDIF           ┘
        ENDM
```

この定義で注意するところは，普通であれば，

```
        IF       SW=1
        JP       L1
        ENDIF
          ~
```

としたいところであるが，先にも解説したようにMACRO-80の場合には，条件付き擬似命令の式フィールドには，パラメータは記述できるが，条件式を記述することはできない．それゆえ，この命令の使用法のテクニックとしては，

> パラメータを真と偽に判定できる形に加工する．

ということになる．それゆえ，先の「CASEマクロ」の場合には，スイッチの状態をチェックするために，擬似命令の式フィールド(パラメータ)を，

(SW) マイナス (チェックしたいスイッチ番号)

という形で，判定したい条件を満たしたときに，偽(ゼロ)になるように工夫されている．それゆえ，使用する条件付き擬似命令もパラメータ(式フィールド)が偽になったときに処理を行なう機能をもった「IFE擬似命令」を使用している．このように，MACRO−80の条件付き擬似命令を使用する場合には，一般的な論理を逆にして条件判断を行なわなければならない．そのために，コメントなどでその内容を明示しておかないと，デバッグ時に思わぬミスを犯すことがある．

(B) アセンブルの過程を調べる例 (IF1, IF2)

この命令は使用法を工夫すればかなりおもしろい使い方もあるかもしれないが，いずれにせよプログラムの本質的なものではなく，デバッグに使用するものとなる．ここでは，アセンブルのパス1とパス2のどちらで生じたエラーかをチェックするために，アセンブル中に，これを表示するためのものを次に示す．なお，この命令は完全に機能が固定されていてパラメータはない．

```
IF1
.PRINTX      *   Pass-1   Now   *
ELSE
.PRINTX      *   Pass-2   Now   *
ENDIF
```

リロケータブル•アセンブラの場合には，外部参照記号の取り扱いに関してパス1とパス2とで，異なる種類のエラーメッセージを表示することがあり，これは，パス1のエラーが，パス2での新しいエラーの原因になっていることが多い．そのようなときに，上記の処理をプログラムの最初の部分に記述しておくことにより，どちらのパスで生じたエラーかを画面上でチェックすることができる．

(C) シンボルが定義されているかどうかを調べる例(IFDEF, IFNDEF)

この命令は，そのプログラム内でパラメータとして指定したシンボルが定義されているか否かをチェックするものである．一般のプログラムであれば，そのプログラムで使用するシンボルというのは定義されているのがあたりまえで，定義されていなければ当然エラーとなるわけであるが，マクロ•アセンブラの場合には，マクロ定義されている部分は必ずしもオブジェクト•プログラムに反映されているとは限らない．すなわち，マクロ定義内で定義されたようなシンボルの

場合には，そのマクロ定義が呼び出されて，その部分が展開されて初めてそのシンボルが定義されることになる．それゆえ，この命令の存在価値があるわけである．これは，マクロ・アセンブラの使用法に慣れていないと，なにかアセンブラのエラーチェック機能の一部のような錯覚をしてしまう命令であるが，上手に使用すると実に効果のある命令ということができる．もちろん，この命令をデバッグ用に使用することもできる．その場合には，「IF 擬似命令」と組み合わせてデバッグ終了時には，これらの部分が展開されないようにしておくとよい．では，マクロ定義内でのより積極的な使用法の例を示そう．

CP/M でディスクの操作を行なう場合，ディスクとのデータのやり取りのためのバッファ領域としての DMA エリアや，ファイル仕様を記録するための FCB エリアが必要になる．これらは，扱うファイルが1つの場合には，5CH からのディフォルト・エリアを使用すればよいが，複数のファイルを扱うような場合には，ユーザがこれらを別のエリアに設定しておかなければならない．これを行なうためのマクロ定義を考えてみよう．

まず，基本的なこのエリアの構造を定義してみる．

```
FCB:
DRIVE:      DS      1       ; ドライブ番号
FNAME:      DS      8       ; ファイル名
FTYPE:      DS      3       ; ファイル名拡張子
EX:         DS      1       ; カレント拡張ナンバー
            DS      2       ; システム予約
RC:         DS      1       ; レコードカウンタ
            DS      16      ; システム予約
CR:         DS      1       ; カレントレコードナンバー
R0:         DS      1       ; ランダムレコードナンバー
R1:         DS      1
R2:         DS      1
DMA:        DS      128     ; データバッファ
```

このようなエリアを番号付きで(例えば，FCB1 というように)自動的に設定するようなマクロ定義は，次のようになる．

```
FCBDEF      MACRO   NO
FCB&NO&:
DRIVE&NO&:  DS      1       ; ドライブ番号
FNAME&NO&:  DS      8       ; ファイル名
```

```
FTYPE&NO&:DS        3          ; ファイル名拡張子
FX&NO&:     DS      1          ; カレント拡張ナンバー
            DS      2          ; システム予約
RC&NO&:     DS      1          ; レコードカウンタ
            DS      16         ; システム予約
CR&NO&:     DS      1          ; カレントレコードナンバー
RO&NO&:     DS      1          ; ランダムレコードナンバー
R1&NO&:     DS      1
R2&NO&:     DS      1
DMA&NO&:    DS      128        ; データバッファ
```

このようなマクロ定義を行なっておき，「FCBDEF 2」というようなマクロ呼び出しを行なうと，メモリ上に，

```
FCB2:
DRIVE2:     DS      1
             ≀
DMA2:       DS      128
```

というような展開が行なわれる．しかし，このマクロ定義では，これを2回以上呼び出すとシンボルの二重定義エラーを生じてしまう．もちろん，このような作業領域は最初に一度だけ定義しておくように習慣付ければよいのであるが，膨大なプログラムになってしまった場合，いちいち，これが行なわれているかどうかをチェックするよりも，ディスクを操作する場合には，必ず使用するFCBをセットするというようなルールにしておくほうが，モジュール別に開発を行なうような場合には，デバッグの効率もよくなる．しかし，先のマクロ定義では，1回目の呼び出しの場合には問題ないが，2回目からの呼び出しでは，エラーが生じてしまう．

そこで，次のようなマクロ定義を使用する．

```
FCBSET      MACRO       NO
            IFNDEF      FCB&NO          ←FCBnのチェック
FCB&NO&:
DRIVE&NO&:DS            1               ←先の FCBDEF の部分
             ≀           ≀
```

```
DMA&NO&:  DS        128
          ENDIF
          ENDM
```

このようにマクロ定義をすることによって，まず「FCBn」というシンボルが定義されているか否かをチェックして，定義されていなければ「FCBDEF」の部分にパラメータを渡してこれが展開される．また，定義されていればこのマクロは何も展開されない．このようにしておけば，シンボルの二重定義ということは起こらないわけである．

では，上記のようなマクロ定義をしておいて，次のプログラムをアセンブルしてみよう．

≪ソース・プログラム≫

```
FCBSET    MACRO     NO
          IFNDEF    FCB&NO
FCB&NO&:
              ≀                    マクロ定義部
DMA&NO&:  DS        128
          ENDIF
          ENDM
          FCBSET    1
          FCBSET    1              マクロ呼び出し部分
          FCBSET    2
          END
```

≪アセンブル・リスト（マクロ呼び出し部分）≫

```
          FCBSET    1
          FCBSET    1
          FCBSET    2
          END
```

なんと１行もマクロ展開されていない！！！　ところが，このリストの最後の部分を見ると，

```
Symbols:
0000'     FCB1      00A3'     FCB2
No Fatal error(s)
```

すなわち，アセンブルは正しく行なわれていたのである．しかし，

```
            FCBSET    1
+     FCB1 :
+
           ~
+
+     DMA1: DS        128
            FCBSET    1
            FCBSET    2
+     FCB2 :
+
           ~
+
+     DMA2: DS        128
            END
```

というリストを期待したことであろう．ところが正しくアセンブルはされてもリストは出力されないのである．これは，パス1とパス2でのアセンブルの相違によるもので，リストが生成された後にこのマクロの展開が行なわれたものと考えられる．

これに関して，マニュアル中に以下の説明がある．

> （MICROSOFT MACRO-80 utility software manual p.79より抜粋）
> 条件ステートメント中のアーギュメント（パラメータ……著者注）はパス1で解決できるものでなければなりません．（中略）IFDEF，または IFNDEF のアーギュメント中のシンボルが IFDEF，または IFNDEF 以降で定義されている場合，パス1では未定義となりますが，パス2で定義済みとなります．

このことから正しいリスト表示が得られないものと思われる．さらに，このパス1とパス2の関係というのは難しいものがあり，先の例を2つのマクロ定義に分けて，

```
FCBDEF          MACRO      NO
FCB&NO&:
                ~
DMA&NO&:        DS         128
                ENDM
```

```
FCBSET          MACRO       NO
                IFNDEF      FCB&NO
                FCBDEF      NO
                ENDIF
                ENDM
```

というように，FCBSET の中で FCBDEF を入れ子にしてマクロ呼び出しすると，アセンブル時に P エラー(フェーズ・エラー)を生じてしまう．このように**論理的には正しいものであっても，アセンブルのパス1とパス2の関係から正しくアセンブルできないものもある**ので注意しなければならない．

(D) パラメータが NUL か空白かどうかを調べる例(IFB，IFNB)

この擬似命令の使用法としては，マクロ呼び出しからのパラメータの数によってマクロ展開の内容を換えるような場合に使用するとよい．例えば，BASIC の「MID$ 関数」を考えてみると，

MID$ (変数名，P1)………… 変数名で示される変数の P1 以降の文字列を取り出す．
MID$ (変数名，P1, P2)……変数名で示される変数の P1 から P2 個の文字列を取り出す．

というように，パラメータが2つの場合と，3つの場合では機能が異なっている．このような場合に，この擬似命令を使用して次のようにマクロ定義することによって，マクロ名は同じでも，与えるパラメータの数によって異なる機能を実現させることができる．

```
MID$        MACRO       VAR, P1, P2
            LD          HL, VAR      ;;変数のアドレス設定
            LD          B, P1        ;;開始位置設定
            IFB         P2
            CALL        MIDSUB1      ;;MID$(A$, 2)型のサブルーチン・コール
            ELSE
            LD          C, P2        ;;文字数設定
            CALL        MIDSUB2      ;;MID$(A$, 2, 5)型のサブルーチン・コール
            ENDIF
            ENDM
```

注） 外部に MIDSUB1, MIDSUB2 というサブルーチンが定義されているものとする．

この定義の展開例を次に示す.

```
        MID$      MOJI$, 5
+       LD        HL, MOJI$
+       LD        B, 5
+       CALL      MIDSUB1   ←条件文によりアセンブルされた部分
        MID$      MOJI$, 2, 3
+       LD        HL, MOJI$
+       LD        B, 2
+       LD        C, 3      ←┐
+       CALL      MIDSUB2   ←┘条件文によりアセンブルされた部分
```

このように，この擬似命令を使用することによってマクロ定義にインテリジェンス(？)をもたせることが可能になる.

(E) 文字列が等しいかどうかを調べる例(IFIDN, IFDIF)

この擬似命令は条件付き擬似命令の中でも最も利用価値の高いものであろう．ディジタル・リサーチ社のアセンブラならば，「IF 擬似命令」を使用して行なう処理であるが，MACRO−80の場合には独立した命令語として用意されている．ところが，この命令語の綴りがきわめてわかりにくく，真の条件判断と偽の条件判断を著者はしばしばまちがえる．そこで，この擬似命令のフォーマットをディジタル・リサーチ社のものと比較してまとめてみよう.

≪ディジタル・リサーチ社のもの≫	≪MACRO−80 のもの≫
IF 'ストリングス 1'= 'ストリングス2'	IFIDN <ストリングス1>,<ストリングス2>
IF 'ストリングス 1'<>'ストリングス2'	IFDIF <ストリングス1>,<ストリングス2>

ディジタル・リサーチ社のものに比較して MACRO−80のものがいかにわかりにくいかがよくわかると思う．しかし，これはいたしかたないことなのでコメントで補うほかないであろう．ではこの擬似命令の使用例を示すことにしよう．ただし，この擬似命令は先にも述べたように非常に利用価値が高く例をあげればきりがないが，この一例から使用法のテクニックを学んでほしい.

例として，汎用の加算命令を考えてみよう．Z80の場合には1バイトの加算を行なう場合，一方のデータは必ずAレジスタに設定されていなければならない．すなわち,

「ADD　B, C　　;B　←　B　+　C」

というような加算は行なえないわけである．これをマクロ定義を利用して可能にしてみよう．これを「拡張加算命令　@ADD」という名前にする．そのためのマクロ定義は次のようになる.

```
@ADD    MACRO   P1, P2
        IFIDN   <P1>, <A>
          ADD   P1, P2
        ELSE
          PUSH  AF
          LD    A, P1
          ADD   A, P2
          LD    P1, A
          POP   AF
        ENDIF
        ENDM
```

要は，第1オペランドをチェックして，それがAレジスタであれば何の細工もしないし，それ以外のレジスタの場合には，まずAレジスタを退避しておいて，指定された第1オペランドの内容をAレジスタに移す，そして基本命令の「ADD　A, r」を使用して加算を行ない，その結果を指定された第1オペランドのレジスタにもどし，Aレジスタを復帰させておく，という簡単なものである．では，使用例のアセンブル・リストを示そう．

```
        @ADD    A, H
  +     ADD     A, H    ←  基本命令のまま
        @ADD    B, C
  +     PUSH    AF    ┐
  +     LD      A, B  │
  +     ADD     A, C  │ 拡張された部分
  +     LD      B, A  │
  +     POP     AF    ┘
```

このようにきわめてスマートに基本命令の拡張を行なうことができる．しかし，この例ではAレジスタの状況にかかわらずAレジスタの退避を行なうためにPUSH, POPを行なっている．もしマクロを使用せずにこのような処理を行なう場合には，Aレジスタの使用状況を見てこの処理を行なうはずである．すなわち，その時点でAレジスタを使用していなければ，この退避という処理を行なう必要はないわけである．ところが，先のマクロ定義では無条件にAレジスタの退避を行なっている．これではメモリの使用量からも，またスピードの点でも無駄が出てくる．そこでこの「@ADD」という拡張加算命令にもう1つオプションのパラメータSを付けてみよう．このパラメータの機能は，これが付いていた場合にだけAレジスタの退避を行なうというルー

ルにしておく．この機能を盛り込んだマクロ定義は，次のようになる．

```
@ADD      MACRO    P1, P2, SW
          IFIDN    <P1>, <A>
            ADD    P1, P2
          ELSE
            IFB    SW
              LD     A, P1
              ADD    A, P2      SW が付けられなかった場合の処理
              LD     P1, A
            ELSE
              PUSH   AF
              LD     A, P1
              ADD    A, P2      SW が付けられた場合の処理
              LD     P1, A
              POP    AF
            ENDIF
          ENDIF
          ENDM
```

すなわち，IFIDN 擬似命令の中でさらに IFB 擬似命令を使用したわけである．このように定義することにより，@ADD 命令の第3パラメータに何か(IFB 擬似命令を使用しているので使用する文字に指定はない)を付加することにより A レジスタの退避が行なわれ，第2パラメータまでだと退避は行なわれない．使用例は，次のようになる．

```
        @ADD     B, C      ←第 2 パラメータまで
   +    LD       A, B
   +    ADD      A, C
   +    LD       B, A
        @ADD     B, C, S   ←第 3 パラメータあり
   +    PUSH     AF        A レジスタの退避
   +    LD       A, B
   +    ADD      A, C
   +    LD       B, A
   +    POP      AF
```

マクロ命令を使用する場合，どうしてもソース・プログラムに比べてオブジェクト・プログラムの大きさが大きくなってしまうので，マクロ定義を行なうときに上記のような配慮をした定義をするように心掛けたほうがよい．

2.5.3　オープンマクロとクローズドマクロ

これまでに解説してきたマクロ定義はすべてオープンマクロと呼ばれるもので，これらのマクロ定義は，呼び出されるたびに同じ定義部分が使用される．小さなマクロ定義であればこれは問題ないが，大きなマクロ定義でパラメータの設定部分と実際の機能部分が分けられるものの場合，呼び出されるたびに実際の機能部分を展開していたのでは，オブジェクト・プログラムの量が莫大なものになってしまう．これを防ぐために「クローズドマクロ」という手法が使用される．このクローズドマクロという手法は，マクロ定義がどのようにアセンブルされるかということをよく知っていないと使用できない．このことに関しては，「2.3 マクロ定義のアセンブルのされ方」を参考にしてほしいが，一言でいうと，マクロ定義の中で定義されているマクロ定義は，その外側のマクロ定義が呼び出された時点で初めてマクロ定義テーブルに登録されるという性質を利用したものである．例えば，

```
VAL     MACRO   P1
        LD      A, P1  ← パラメータ設定
        LD      B, A        ┐
        AND     11110000B   │
        RRCA                │共
        LD      C, A        │通
        RRCA                │部
        RRCA                │分
        ADD     A, C        │
        LD      C, A        │
        LD      A, B        │
        AND     00001111B   │
        ADD     A, C        ┘
        ENDM
```

というマクロ定義を考えてみよう．このプログラムは，P1で与えられた1バイトの BCD データを1バイトの2進データに変換するものであるが，呼び出されるたびに変化するのは最初のパラメータ設定部分だけである．これを呼び出されるたびにすべて展開していたのではメモリの

無駄使いは目に見えている．一般的には，この共通部分をサブルーチンにしてマクロ外部に置き，マクロ定義内ではこれをコールするという方法をとるのが普通であろう．しかし，このクローズドマクロという手法を取ると，これと同じことがマクロ定義内で行なえるのである．

```
VAL        MACRO      P1               ┐
           JR         J1               │  外側のマクロ
 VALSUB :                              │
           LD         B, A             │
           AND        11110000B        │
           RRCA                 共      │
           LD         C, A      通      │
           RRCA                 部      │
           RRCA                 分      │
           ADD        A, C             │
           LD         C, A             │
           LD         A, B             │
           AND        00001111B        │
           ADD        A, C             │
           RET                         │
         J1:                           │
  VAL      MACRO      P11              │
           LD         A, P11 ←パラメータ設定   内側のマクロ
           CALL       VALSUB           │
           ENDM                        │
           VAL        P1               │
           ENDM                        ┘
```

マクロ定義の内部で，また同じ名前でマクロ定義をしているという複雑なマクロ定義であるが，これを実際にマクロ呼び出しをした展開例から示そう．

```
           VAL        99H       ←マクロ呼び出し１回目
+          JR         J1
+  VALSUB:
+          LD         B, A
           ~
```

```
+           RET
+       J1:
            ENDM
+           LD          A, 99H
+           CALL        VALSUB
            VAL         11H       ←マクロ呼び出し2回目
+           LD          A, 11H
+           CALL        VALSUB
```

このように1回目の呼び出しのときと，2回目以降の呼び出しの場合では，展開される内容が異なっている．これにより無駄な部分が何度も展開されることなく効率のよいマクロ展開が得られる．なぜこのような展開が行なわれるかというと，アセンブラがマクロ定義をチェックする時点では，外側のマクロ定義をマクロ定義テーブルに登録する．そして内側のマクロ定義は，その内部に記録されマクロ名はマクロ定義テーブルに登録されない．それゆえ，1回目にこのマクロが呼び出された場合には，外側のマクロが呼び出しの対象になり全体が展開される．このときに，内側のマクロ定義がマクロ定義テーブルに登録される．この内側のマクロ定義は，外側のマクロ定義よりもマクロ定義テーブル上で後ろに位置するために2回目以降の呼び出しでは，この内側のマクロ定義が呼び出されることになる(マクロ定義テーブルは，後方から調べられる)．これがクローズドマクロの原理となっているわけである．このクローズドマクロを定義する場合の方法をもう一度まとめてみよう．マクロ名は，仮に TEST としておく．

```
      TEST      MACRO       パラメータ
同              LOCAL       J1 ←他のマクロ定義のことを考えて二重定義を避
じ                              ける
名              JR          J1
前    TESTSUB:
に              サブルーチン部分       1回目だけ展開される部分
す              (処理本体)
る        J1:
      TEST      MACRO       パラメータ

                パラメータ設定部分     2回目以降の展開される部分
                CALL        TESTSUB
```

```
ENDM
TEST        パラメータ← 2回目以降のマクロ定義の登録と呼び
                       出し
ENDM
```

この手法は，他のモジュールとリンクしないようなプログラムの場合にはきわめて効率のよいものであるが，リロケータブル•マクロアセンブラを使用してモジュール開発を行なう場合には2.6節で解説するような手法を用いたほうがよいであろう．

2.5.4　デバッグに関するテクニック

マクロ定義を利用したプログラムでは，そのデバッグの方法もいくつかのテクニックが必要になる．それは，マクロを利用した場合，それが呼び出されてマクロ展開された時点でソース•プログラムが確定されるわけで，アセンブル以前のソース•リストを見ただけではどのようなプログラムになるのかを判断するのは難しい．というよりも，マクロを利用したプログラムのバグというのは，このマクロ展開が期待どおりになっていないことから生ずるもので，マクロ定義を利用したプログラムのデバッグとは，アセンブル•リストから始まると考えてもよい．ともかく一旦アセンブルしてみないと，マクロ呼び出しを行なった部分がどのように展開されるかわからないわけで，以下に，マクロ定義に関する基本的なデバッグのテクニックを解説する．

（A）　アセンブル•リストに関するテクニック

アセンブル•リストに関するテクニックとして，「.LALL 擬似命令」を使用するということがあげられる．一般にソース•リスト中で何の指定もしないと「.XALL擬似命令」を指定したのと同じことになり，アセンブル•リストにはオブジェクトコードを生成した部分のみが表示される．それに対して「.LALL 擬似命令」を付けておくとすべてのマクロ展開部分が表示される．それゆえ，特にチェックしたい部分の前にこの「.LALL 擬似命令」を付け，必要がなくなったところで「.XALL 擬似命令」を付けるようにすると，効率のよいアセンブル•リストを取ることができる．

（B）　デバッグしやすいプログラムのつくり方に関するテクニック

このマクロ•ライブラリ手法を用いてプログラム開発を行なう場合には，マクロ定義部分が受け持つ役割というのは，外部のリロケータブル•モジュールへのパラメータの受渡しが主な機能になるわけであるが，その場合，次の2つのケースが考えられる．

- すでに開発済の外部モジュールとのインターフェースとしてマクロを定義する場合．
- 外部モジュールとなる部分も含んでマクロを定義する場合．

前者の場合には，外部モジュールの機能はすでにデバッグ済であろうから，単に外部モジュールが要求している形にパラメータが設定されているかどうかをチェックすればよい．この場合，外部モジュールまで含めてデバッグすると，リンク時にこの外部モジュールライブラリをリンクしなければならず，リンクに要する手間を考えるとこの外部モジュールと同名のダミーサブルーチンを同じプログラムに記述しておきチェックを行なった後，実際にリンクして最終テストを行なうとよい．後者の場合には，このダミーサブルーチンの部分が実際の機能部分となり，デバッグの対象となる．そして，デバッグ完了後にこの部分だけを別ファイルにしてアセンブルし，ライブラリマネジャを使用してライブラリ・ファイルに登録しておけばよいわけである．ただし，この場合，モジュール名を“共通参照記号(PUBLIC 宣言)”として宣言しておくことを忘れてはならない．また，前者，後者いずれの場合も，マクロ定義部に関しては，やはり別ファイルにしておき，エディタのテキストファイルの読み込み機能，または，PIP ユーティリティによりマクロ定義ライブラリに登録しておけばよい．この場合には，そのマクロ定義内で使用している外部モジュール名を“外部参照記号(EXTRN 宣言)”として宣言しておくことを忘れてはならない．またいずれの場合も，デバッグ用に使用した「IF 擬似命令」を削除することも必要である．

この形のテストプログラムの構造は次のようになる．テスト用フラグと「IF 擬似命令」の使用法に注意してほしい．

```
TEST      EQU       0FFH      デバッグ用フラグ
-------------------------------------------------------------
          ≪マクロ定義部≫
ZZZ       MACRO     *****
          IFE       TEST   ┐  デバッグフラグがゼロのとき
            EXTRN   EXSUB  │  アセンブルされる
          ENDIF  ──────────┘
           ∫
          CALL      EXSUB ←外部モジュール名
           ∫
          ENDM
-------------------------------------------------------------
          ≪テストスタブ≫
          ZZZ       *****     パラメータの授受と結果を確認しやすいよう
           ∫                  に考える
-------------------------------------------------------------
          ≪サブルーチン部分≫
          IF        TEST
```

```
EXSUB:
            ～          ┐
                        │ デバッグフラグの状態により展開される
            RET         ┘
            ENDIF
```

(C)　デバッグに関する一般的テクニック

ときには「IFDEF 擬似命令」のところで例に示したプログラムのように，論理的には正しくても，アセンブルの過程でエラーになってしまうこともある．これらのバグは，バグというよりも，そのアセンブラの処理方法に関係するもので，アセンブラの作成法を相当に熟知していて，そのアセンブラの限界を知っていないと理解できない．また，中にはアセンブラ自体のバグではないかと思うことにも著者自身何度か遭遇したが，これらのバグの規則性を見つけることはかなり困難で，著者の未熟さゆえのプログラミングミスか，それとも本当にアセンブラ自体のバグであるのか，本書で解説するまでには至らなかった．これらの解析不能なバグは，主にマクロの入れ子を行ないマクロのネスティングレベルが深くなったときに起こるようである．特に入れ子の中での「EXITM 擬似命令」には十分に注意したほうがよいだろう．

また，これはマクロ定義そのものに問題があり，作成者の明らかなミスであるが，マクロ名を不当に定義してしまった場合，アセンブルの途中で暴走してしまうことがある．

これはどのようなときに起こるかというと，次に示すようにマクロ展開が無限ループに入ってしまったときに起こることが多い．

```
ADD     MACRO     P1, P2
        IFIDN     <P1>, <A>
          ADD     P1, P2    ←┐
        ELSE                 │ これらの部分が展開されると再び ADD
          PUSH    AF         │ マクロが展開され，さらにその中で展開
          LD      A, P1      │ されるという無限ループができてしまう
          ADD     A, P2     ←┘
          LD      P1, A
          POP     AF
        ENDIF
        ENDM
```

この例の場合には暴走せずに“?Stack overflow, try more P switches”というメッセージを表示してアセンブルを終了してしまう．このような場合，アセンブル・リストも出力されないの

で本当にプログラムが大きすぎてスタックエリアが足りなくなったのか，それともバグなのかが非常に判断しにくい．

このような場合には，これらの不当なマクロ名による「マクロの無限ループ」をチェックすることもデバッグのテクニックとなる．

これらを含めて，難解なバグを退治するときのチェックポイントをあげると次のようになる．

- EXITMを使用した場合，これを使用しなくてもよいような論理に変換してみる．
- マクロの入れ子がしてある場合には，これをはずしてみる．マクロの入れ子はフェーズエラーを起こすことが多い．
- マクロ定義内の局所的なシンボルがLOCAL定義されているかチェックする．
- マクロ定義内のブロック擬似命令に対するENDM擬似命令が正しくなされているかチェックする．
- 条件付き擬似命令とELSE，ENDIFは正しく対応しているかチェックする．
- マクロ定義内の条件付き擬似命令の内部からEXITM擬似命令で脱出してないかチェックする．
- パラメータに対するマクロ定義用演算子の使用法をチェックする．
- パラメータに対する要求が満たされているかチェックする．絶対的な値をもつパラメータしかとれない擬似命令もある．

2.6　マクロ・ライブラリ作成の注意点

リロケータブル・マクロアセンブラを利用してマクロ・ライブラリを作成する場合，「2.1 マクロ・ライブラリの有効性」で解説したようにマクロ定義部と，その中で使用するモジュール（サブルーチン）とは別のファイルになる．すなわち，マクロ定義部はアセンブル時に“INCLUDE 擬似命令”などによってソース・プログラムに結合されるテキスト・ファイル（ASCII ファイル）となるし，その中で使用されるモジュールは，個々にアセンブルされたリロケータブル・オブジェクト・ファイルとなり，リンク時に“.REQUEST 擬似命令”などによりリンクされる．この関係を図示すると**図2.2**のようになる．

このことから，「マクロ定義ライブラリ」，「外部モジュール・ライブラリ」，「アセンブル対象プログラム（メイン・モジュール）」では，次にあげる要素を満足していなければならない．

≪マクロ定義ライブラリ≫

- マクロ名が他のシンボルと重複しないこと．
- 使用している外部モジュール名を外部参照記号として登録すること．
- 定義内部で使用しているシンボルは必ず LOCAL 指定をしておくこと．
- INCLUDE 擬似命令を使用しないこと（INCLUDE の入れ子はできない）．
- ライブラリは機能別にいくつかのファイルに分けられていたほうがよい．
  （このライブラリはソースファイルに結合されるので，あまり1つのライブラリが大きいとマクロ定義テーブル用のメモリが不足したり，アセンブルに時間がかかるようになる）

≪外部モジュール・ライブラリ≫

- モジュールの単位はなるべく小さいほうがよい．
- モジュール名は共通参照記号として登録してなければならない．
- パラメータの授受用以外のレジスタはその内容が保証されなければならない．
- スタックのレベルなどに変化を与えてはならない．

≪メイン・モジュール≫

- プログラムの先頭で必要となるマクロ定義ライブラリを「INCLUDE 擬似命令」によって定義しておく．
- 必要となる外部モジュールを含むライブラリを「.REQUEST 擬似命令」によって定義しておく．

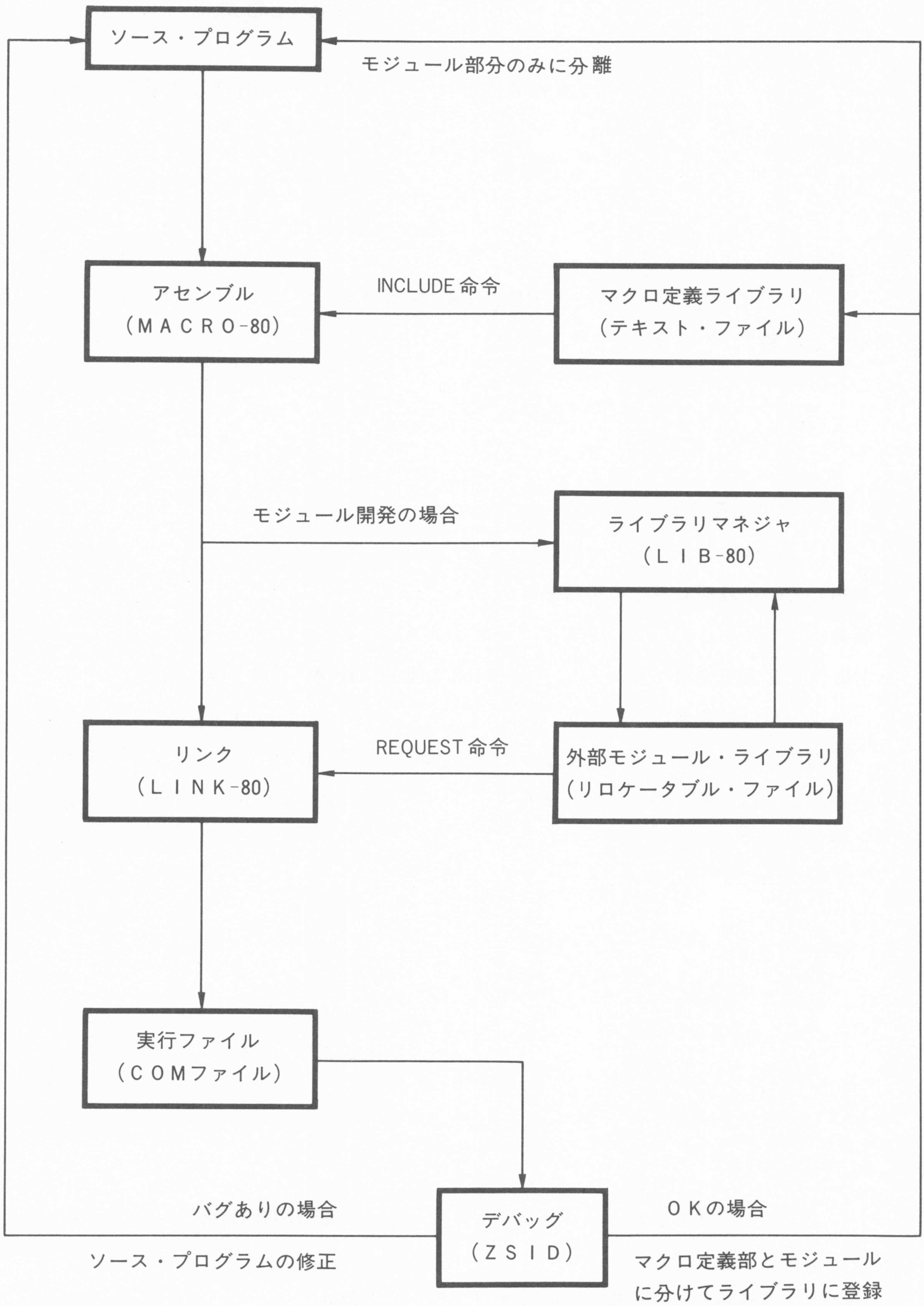


図 2.2 マクロ・ライブラリ作成のプロセス

第3章

ソフトウェア・ツールの種類と効果的な使用法

[この章のポイント]

リロケータブル・マクロアセンブラを使用したプログラム開発に利用できるソフトウェアツールの効果的な使用方法を，次の6つのツールについて解説する．

◎ テキスト・エディタ (ED, WORD-MASTER, WORD-STAR)
◎ リロケータブル・マクロアセンブラ (MACRO-80)
◎ クロスリファレンス・ユーティリティ (CREF-80)
◎ リンクローダ (LINK-80)
◎ ライブラリマネジャ (LIB-80)
◎ シンボリック・デバッガ (ZSID)

3.1　エディタの種類と使用法

ここで解説するエディタというプログラムは，ソース・プログラムの作成をするときに使うもので，キーから入力したプログラムを，ディスケットにアスキー形式のファイルとして書き込むもので，テキスト・エディタと呼ばれている．

したがって，アセンブラでプログラムを作成するような場合以外にも(例えばコンピュータを使って他の言語でプログラムを作成したり，または文章をタイプするなど)，よく使われている．

テキスト・エディタにもいろいろ種類があってそれぞれ特徴があるが，CP/M で使用するテキスト・エディタで有名なものは，標準装備のキャラクタ・エディタであるED，また，キャラクタ・エディタの機能とスクリーン・エディタの機能をもったWord-Master，英文用ワード・プロセッサーとしての機能を十分に装備したWord-Star，の3種類をあげることができる．その中でもCP/M で最もよく使用されている，Word-Master を中心に取り上げて，使用法や他のテキスト・エディタ(ED やWord-Star)との違いなどについて次の手順で解説していくことにする．

3.1.1　ED，Word-Master，Word-Star の特徴
3.1.2　エディタの基本機能
3.1.3　テキスト修正の方法
3.1.4　文字列の検索と置換
3.1.5　テキスト・ファイルの Read・Write

3.1.1　ED, WORD-MASTER, WORD-STARの特徴

(A)　EDの特徴

EDは，以前コンピュータをTSSで使っていたテレタイプ・ライタ用のエディタによく似ており，キャラクタ・エディタと呼ばれているエディタである．CP/M には，他のDDTやASMなどと一緒にEDが基本システム・ソフトとして付いてくる．

したがって，パーソナル・コンピュータを使い慣れている人には，スクリーン上でテキストを確かめながら修正できるBASICのスクリーン・エディタのほうがはるかに使いやすいであろう．しかし，エディタの基本的な機能と動作を理解するためには，EDのほうがシンプルにできているので適しているといえる．

EDの機能としては，本来のエディタの使用目的である，

- 新規テキスト・ファイルの作成.
- 旧テキスト・ファイルの修正.

の2つの他に，

- ライブラリ・ファイルからのプログラムの読み込み.
- 同一コマンドを繰り返し行なわせるマクロ機能.
- 文字列の置き換えを行なう機能.

などの便利な機能が準備されており，2，3日使ってEDに慣れてしまえば手離せなくなってしまう．ただ先に述べたようにEDはキャラクタ•エディタなので，修正位置をキャラクタ•ポインタ（CP）で示す形式を取っているため，たえずテキストの全体がどのようになっているのかを頭の中で把握しておかないと手に負えなくなってしまう可能性が大きい.

このようなことを避けるためには，事前にプログラムのコーディングをしっかりとやっておくようにし，なおかつ画面に何行かを表示させて，確認しながらテキストの作成を行なうことである.

「注意一秒，ケガ一生」，「ころばぬ先の杖」

という気持ちで，多少遠回りのような気がしても気をつけることにこしたことはない.

（B） WORD−MASTERの特徴

WORD−MASTERは，マイクロプロ社が作成・発売したスクリーン•エディタである．その機能は，多くのパーソナル•コンピュータで使用されているエディタと同じタイプのもので，テキストを画面に表示させながら自由に修正箇所を示すカーソルを移動させることができるものである．したがって，**カーソル移動が可能な機種で使うのが，本来の用途に合った使い方**といえる．では，そのような（カーソル•アドレッシングが可能な）機種でしか使えないのかというとそうではなく，前述のEDと同じようなキャラクタ•エディタ的な使い方もできるようになっている．ただし，カーソルの移動のためのコードがメーカによって異なるので，汎用性をもたせるためにコントロール•キーを使った方法で，カーソルの移動をサポートしている．そのため使い慣れるまで大変なところもある．その点WORD−STARでは，カーソル移動のコードも使用するパーソナル•コンピュータに合わせて変更（このことをインストールするという）することができるので，非常に使いやすいものとなっている（WORD−MASTERは，発売元のマイクロプロ社以外ではインストールできない）.

WORD−MASTERの機能を，前述のEDと比べると，

- スクリーン・エディットが可能なビデオ編集モード.
- ファイルの大きさを気にせずに編集ができる.
- ライブラリ・ファイルの読み込みと作成ができる.
- 操作方法が参照・確認できるヘルプ機能.

などの優れた機能が準備されているのが特徴である.

このビデオ・モードによるテキスト作成の操作性のよさは，文字には表わしにくいが，実際に使用してみればその相違がすぐにわかるであろう．また，EDの場合だと作業領域に入り切らないようなテキストを作成したり編集したりする場合には，テキストをいくつかに分割して，ディスクに書いたり読んだりの作業を行なうようにしなければならないのが，このWORD−MASTERでは全く気にする必要がなく，のんびりと行なうことができる．さらにプログラムの作成を進めているときに，後で使えそうな部分をテキストライブラリとしてディスクに残しておきたい場合に適当な名称を付けてディスクに登録しておくことが簡単にできてしまう．ライブラリは1回つくっておけば何回でも使えるので，タイプ・ミスの減少や同じルーチンを何回も考えたりタイプしたりする手間と時間・労力の節約になる.

また，操作方法などは一旦憶えてしまえばほとんど問題ないが，人間稼業を長くやっているうちにはど忘れということもある.

そのようなときに，ヘルプ機能を使えば即座に操作方法を画面に出して説明してくれるというきわめて重宝な機能も準備されている.

(C) WORD−STARの特徴

WORD−STARは，前述のWORD−MASTERと同じマイクロプロ社の製品であるが，これまでのEDやWORD−MASTERなどとは異なり，単なるエディタではない．すなわち，パーソナル・コンピュータを英文専用のワード・プロセッサーに変身させてしまうすばらしいスクリーン・エディタなのである．普通のタイプライターだと，一端タイプしてしまえば，後でタイプミスに気が付いてもまちがえた箇所を消して，またその上からタイプし直すという方法しかないが，このWORD−STARを使って文書の作成を行なえば，万一まちがっても，いつでも気がついたときに簡単に直すことができるのである．したがって，WORD−STARは，**英文専用ワード・プロセッサーとして使うのが本来の使い方**といえる．しかし，WORD−STARはワード・プロセッサとして，文書（文書のことを，ドキュメント・ファイルと呼ぶ）の作成に使うだけではなく，テキストファイル（文書以外のものを，ノン・ドキュメント・ファイルと呼ぶ）の作成にも使うことができる.

WORD−STARの機能を，これまでのEDやWORD−MASTERと比べると，

- 各種のコントロール・コードが自由に定義できる.
- 文字列を確認しながら，検索や置き換えができる.
- 文書の印字のとき，強調印字や下線などの指定ができる.

などの点で優れている．したがって，ED や WORD-MASTER などでは使えなかった，最近のパーソナル・コンピュータにはつきものの，カーソル移動キーやインサート，デリートなどのキーを，名称どおりの機能に使うことができる(ただし，ファンクション・キーは使えない)．また，ED や WORD-MASTER では文字列の置き換えが単純で，置き換えの必要のない部分までも置き換えられてしまうという事故がよく起こったが，WORD-STAR では画面上で確認しながら置き換えを行なうことができるので，安心して行なうことができるようになった．さらに WORD-STAR は，MAIL-MERGE と一緒に使うことにより，文書の差し込みや定型文書の処理ができるようになったり，SPELL-STAR と一緒に使うことにより，英単語のスペル・チェックを自動的に行ない，まちがいの箇所を指摘してくれるなどのすばらしい機能が利用できるのも特徴の1つである．ただ，あまりにも機能が豊富なので，WORD-STAR を十分に使いこなせるようになるまでには，相当な時間を要するというぜいたくな欠点をもっているのである.

3.1.2　エディタの基本機能

(A)　エディタの起動法

CP/M で使用するエディタはすべて(ED，WORD-MASTER，WORD-STAR のいずれも)，ディスク上のファイルを対象にする．そのため ED や WORD-MASTER では，エディタを起動するときには対象となるファイル名を指定しなければならない.

例えば，

```
A>WM　B:WORD.TST←——編集するファイル名
  COPYRIGHT (C) 1987  MICROPRO INTERNATIONAL CORPORATION


……MicroPro WORDMASTER release --------
NEW FILE←——新規作成のファイルの場合

■←——画面がクリアされた後，カーソルが表われる
```

<ファイルが新たに作成される場合>

普通ディスクを使うときは，システム専用のディスク（システム・ディスクと呼ぶ）と，作業用ディスク（ワーク・ディスク）とに分けて使用するので，ドライブAをシステム用に，ドライブBを作業用ディスクとした場合について考えることにする．

上記のサンプル画面のように，WM B:WORD.TSTとキー入力した場合には，システム・ディスクであるドライブAに入っている**WM.COM**のプログラム，すなわちワード・マスター（WORD−MASTER）を起動して，ドライブB上に**WORD.TST**という名前のテキスト・ファイルを新しく作成するか，もしくはすでに存在するものを修正することになる．作成と修正のどちらになるかは，ドライブB上に**WORD.TST**というファイルが存在するか，存在しないかによって決まる．

ファイル名が存在しない場合	──────→	新規作成（NEW　FILE）
ファイル名がすでにある場合	──────→	修正作業

では，ファイル名がすでにある場合には，どのような画面に変わるのかというと，

```
A>WM B:WORD.TST
  COPYRIGHT (C) 1987 MICROPRO INTERNATIONAL CORPORATION

……MicroPro WORDMASTER release ---------

  TEST OF WORD MASTER.←─テキストが表示される
  FILE IS ALLREADY EXIST!
```

<ファイルがすでに存在している場合>

のように，すでにドライブBに登録されているファイルを画面に表示して，修正作業のモードになる．どのエディタについてもいえることで，注意しなければならないこととして，ドライブBは作業用のディスクとして使用するのであるから，書き込み禁止になっていては駄目なことである．このようなディスクを使ってエディタを使おうとすると，

Bdos Err On B:

というエラー・メッセージが表示され，エディタを起動することができない．これと同じエラーは，ドライブBのふたを開けたりディスクを入れ替えたりしても起こる．このような場合には，control+Cでドライブのリセットを行なった後で，再びエディタを起動すると正常に動作するようになる．

また，EDを起動した場合は，

```
A＞ED B:WORD.TST
      :*#A←ファイルの入力
      :*#T←ファイルの表示
    1:TEST OF WORD MASTER.
    2:FILE IS ALLREADY EXIST!
```

＜すでにファイルが存在している場合＞

```
A＞ED B:WORD.TST

   NEW FILE←—新規作成の表示

   :*←—プロンプト
```

＜ファイルが新たに作成される場合＞

注）プロンプトとは入力を促す記号のことで，パーソナル・コンピュータでいうカーソルと同じ意味である．

このようにEDの場合は，ファイルがすでに存在している場合でも，画面テキストの表示はされないし，ファイルの内容であるテキストの読み込みも一切行なわれないのである．テキストを読み込むためには，Aコマンド（add command）を使ってディスク上のファイルからテキストを読み込み，Tコマンド（type command）でようやく内容が確認できるようになる．

ワード・スターの場合の起動法は，機能が豊富なだけに，多少手続きを踏んで行なう必要があるので，順を追って説明していくことにする．

◎ WS と入力し改行する

すると次のような画面が一瞬表示され，次に NO-FILE MENU が表示される.

```
MicroPro WordStar release ?.??   serial # ????????
COPYRIGHT (C) 198?  MicroPro International Coporation
```

```
          editing  no file
                    <<<NO-FILE MENU>>>
 --Preliminary Commands--     | --File Commands--  |  -System Commands-
L Change logged disk drive    |                    |  R  Run a program
F File directory  off(ON)     | P  Print a file    |  X  EXIT to system
H set help level              |                    |
--Commands to open a file--   | E  RENAME a file   |  -WordStar Options-
  D Open a document a file    | O  COPY     a file |  M  Run Mailmerge
  N Open a non-document file  | Y  DELETE  a file  |  S   run SpellStar

DIRECTORY of disk A:
  WS.COM   WSMSGS.OVR WSOVLY1.OVR FORMAT.COM PIP.COM
  STAT.COM
```

◎ L と入力する

起動したままの状態では，作業用ディスクがドライブ A になっているので，これを変更するために L と入力する.

◎B：と入力し改行する

作業用ディスクをドライブBに変更するためにB：と入力し改行する．

```
L                    editing no file
  The LOGGED DISK (or Current Disk or Default Disk) is the
  disk drive used for files except those files for which
  you enter a disk drive name as part of the file name.
  WordStar displays the File Directory of the Logged Disk.

THE LOGGED DISK DRIVE IS NOW A:←―立ち上げた状態でのドライブ指定はA

NEW LOGGED DISK DRIVE (letter, colon, RETURN)? B:←―ドライブBに変更

DIRECTORY of disk A:
  WS.COM  WSMSGS.OVR  WSOVLY1.OVR FORMAT.COM PIP.COM
  STAT.COM
```

◎Nと入力する

NO-FILE MENUでは，これから作成するファイルが文書ファイル(document file)か，それともテキスト・ファイル(non-document file)かの指定をしなければならない．今回は，プログラムの作成をする目的でしか使わないのでnon-document fileの指定をするために，Nと入力する．最後に，作成または修正するファイル名をWORD.TSTと入力し改行する．

```
N           editing no file
  Use this command to create and alter program source files
    and other non-documents. Word wrap default off;
    tabbing defaults to fixed (TAB chars in file; 8-col stops);
    page breaks not shown; hi bit flags not used in file.
  For normal word processing uses, use the "D" command instead.


     A file name is 1-8 letters/digits, a period,
```

```
      and an optional 0-3 character type.
      File name may be preceded by drive letter A-D
      and colon, otherwise current logged disk is used.

       NAME OF FILE TO EDIT? WORD.TST ←――ファイル名を入力する
DIRECTORY of disk B:
  WORD.TST
```

ワード・スターの場合も，作成と修正のどちらになるかは，ドライブB上に WORD.TST というファイルが存在するか，存在しないかによって決まり，修正の場合にはすでに存在するファイルの内容が画面に表示されて，エディット・モード（修正可能の状態）に入る．

```
N          editing no file
  Use this command to create and alter program source files
    and other non-documents. Word wrap default off;
    tabbing defaults to fixed (TAB chars in file; 8-col stops);
    page breaks not shown; hi bit flags not used in file.
  For normal word processing uses, use the "D" command instead.

     A file name is 1-8 letters/digits, a period,
     and an optional 0-3 character type.
     File name may be preceded by drive letter A-D
     and colon, otherwise current logged disk is used.

     NAME OF FILE TO EDIT?  WORD.TES ←――新規作成のファイルを指定

NEW FILE ←――新規作成の表示
```

＜ファイルが新たに作成される場合＞

```
             B:WORD. TST   FC=1 FL=1 COL 01
                    <<<  MAIN  MENU   >>>
   --Cursor Movement--    -Delete-    -Miscellaneous-     -Other Menus-
^S char left  ^D char right ^g char    ^I Tab  ^B Reform  (form Main onl
^A word left  ^F word right DEL chr lf ^V INSERT ON/OFF ^J Help ^K Bl
```

```
^E line up    ^X line down  ^T word rt ^L Find/Replce again ^Q Quick ^P Pr
    --Scrolling--        ^Y line   RETURN End paragraph ^O Onscreen
^Z line up    ^W line down           ^N Insert a RETURN
^C screen up  ^R screen down         ^U Stop a command
test of wordstar←――テキストの表示
allready exist!
```

＜ファイルがすでに存在している場合＞

(B) WORD-MASTER と ED の共通コマンド

原則的には，ワード・マスターのコマンド・モードで使うコマンドと，EDのサブコマンド・モードで使うコマンドとは共通しているが，EDのすべてのコマンドをワード・マスターのコマンド・モードで使うことはできない．それでは，参考のためどのようなコマンドが共通して使用可能か，コマンド名と機能をあげ，簡単な例を示しておくことにする．

≪行単位で働く共通コマンド≫
・@　ポインタを1行後方に移動しポインタの位置の行を表示する． @　は，改行することを表わしているので単に改行するだけでよい．
・+-nL　ポインタを+-n行移動する． +3L　なら，3行分ファイルの後方にポインタを移動する． -5L　なら，5行分ファイルの前方にポインタを移動する．
・+-nK　ポインタの位置より，+1n行削除する． +4K　なら，ポインタの位置からファイルの後方に向かって4行削除する． -7K　なら，ポインタの位置からファイルの前方に向かって7行削除する．
・+-nT ポインタの位置より，+-n行表示する． +11T なら，ポインタの位置からファイルの後方に向かって11行表示する． -23T なら，ポインタの位置まで，11行前から表示する．
・+-B　ポインタをファイルの先頭，もしくは最後尾に移動する． +B　なら，ポインタをファイルの先頭に移動する． -B　なら，ポインタをファイルの最後尾に移動する．

≪文字単位で働く共通コマンド≫
・+-nC　ポインタを+-n文字分移動する． +5C　なら，ポインタを5文字，行の後方に向かって移動する． −2C　なら，ポインタを2文字，行の先頭に向かって移動する．
・+-nD　ポインタの位置から+1n文字削除する． +3D　なら，ポインタの位置から，行の後方に向かって3文字削除する． −4D　なら，ポインタの位置から，行の先頭に向かって4文字削除する．
≪ページ単位で働く共通コマンド≫
・+-nP　ポインタを1ページ移動後，続く+-nページを表示する． +2P　続く2ページを表示，ポインタを最後に表示したページに移動する． −3P　前の2ページを表示，ポインタを最初に表示したページに移動する．
≪挿入に関する共通コマンド≫
・I@　ポインタのある行に続けて，新しい行を挿入する． I@　なら，挿入(インサート)モードに入る．
・Iテキスト$　ポインタの後にテキストを挿入する． Iinsert$　なら，ポインタの後にinsertという文字列が挿入される． 注）$はESCキーまたは改行を意味する．

このようにポインタの移動や挿入に関しては，共通に使えるコマンドが多いが，その他のコマンドについては，機能が同じでもコマンドを表わす文字が違う場合や機能自体が違っているものばかりである．しかし，ワード•マスター自体にもいくつかのバージョンがあるので，簡単にはいえないようである．

ワード•マスターのいくつかのバージョンを比較してみると，古いバージョンのものはEDに近く，最近のバージョンのものは，むしろワード•スターのコマンド名に近づいてきている．

(C)　カーソルの移動法

カーソルは，次に入力する文字の位置を表示するもので，点滅する『■』のようなマークで示さ

れる．したがってエディタでプログラムを作成するときには，たえずカーソルの位置を確認しながら入力をするようにしなければならない．このカーソルは，上下左右に1文字ずつ移動するほか，単語ごとや，1行中や1画面中での移動や，ページごとの移動ができる．またパーソナル・コンピュータで最近使われるようになったロール•アップ，ロール•ダウンと同じ機能を，コントロール•キーを使って操作が行なえるようになっている．

（1）　文字単位でのカーソル移動法

1文字単位でカーソルを移動するには，コントロール•キーとD•S•E•Xの4個のキーを使って行なう．

キー操作	カーソルの移動方向
CONTROL+D	1文字分，右にカーソルを移動する．
CONTROL+S	1文字分，左にカーソルを移動する．
CONTROL+E	1行上の行にカーソルを移動する．
CONTROL+X	1行下の行にカーソルを移動する．

注）　ワード•スターの場合も同じ操作になる．

上の表に見られるようにCONTROLキーとDとSのキーによって，カーソルを1文字分左右に移動することができる．ただし，カーソルが移動することができるのは（画面に表示されないものも含めて）文字が実際に存在する位置に限られている．例えば，タブによってスキップしているような所ではカーソルを移動しようとしても受け付けられない．それでは，CONTROL+Eを行なったときに，たまたま上の行がカーソルのある行より短かったような場合はどうであろうか．

```
TEST                                        TEST←カーソル
CURSOR MOVE   ――→  CONTROL+E  ――→    CURSOR MOVE
```

このように，もとあったカーソル位置に一番近い位置でかつ左側の文字にカーソルが移動することになり，結果として行の右端にカーソルが移動する．これと同じ現象が，もとのカーソルのあった位置の上の行がタブによってスキップしている区間に当たるような所でも起こるので注意する必要がある．

（2）　単語単位でのカーソル移動法

単語単位でカーソルを移動するには，コントロール•キーとAとDの2個のキーを使って行な

う．ただし，ここでいう単語とは，アルファベット（A～Z）や数字（0～9）の連続したものという程度のものである．

キー操作	カーソルの移動方向
CONTROL+F	1単語分，右にカーソルを移動する．
CONTROL+A	1単語分，左にカーソルを移動する．

注）ワード・スターの場合も同じ操作になる．

上の表に見られるようにCONTROLキーとFとAのキーによって，カーソルを単語単位で左右に移動することができる．

（3）　1行中でのカーソル移動法

1行中でのカーソルを移動するには，コントロール・キーとBのキーを使って行なう．この行単位のカーソル移動は，次のページ中でのカーソル移動と同じで，非常に特殊な機能として偶数回キー操作を行なうともとにもどってしまうトグル式になっているので，十分注意する必要がある．

キー操作	カーソルの移動方向
CONTROL+Bを奇数回	行の左端にカーソルを移動する．
CONTROL+Bを偶数回	行の右端にカーソルを移動する．

注）ワード・スターの場合は，CONTROLキーとQDとQSという操作になる．

すなわち，移動する前にカーソルが行のどこにあっても関係なく，CONTROL+Lを何回押したかによってカーソルの移動する位置が決まるのである．

（4）　ページ中でのカーソル移動法

ページ中でのカーソルを移動するには，コントロール・キーと^のキーを使って行なう．このページ中でのカーソル移動は，前の1行中でのカーソル移動と同じトグル式なので注意する．

キー操作	カーソルの移動方向
CONTROL+^を奇数回	画面の上端にカーソルを移動する．
CONTROL+^を偶数回	画面の下端にカーソルを移動する．

注）ワード・スターの場合は，CONTROLキーとQEとQXという操作になる．

この場合も移動する前にカーソルが画面のどこにあっても関係なく，CONTROL+∧を何回押したかによってカーソルの移動する位置が決まるのは，前の1行中でのカーソル移動の場合と同じである.

(5)　ページ(画面)単位でのカーソル移動法

カーソルを1ページ前にもどしたり，次のページに移動したりするのには，コントロールキーとCとRの2個のキーを使って行なう.

キー操作	カーソルの移動方向
CONTROL+C	前のページにカーソルを移動する.
CONTROL+R	次のページにカーソルを移動する.

注)　ワード•スターの場合も，同じ操作であるが2/3画面単位の移動となる.

すなわち，CONTROL+Cを押すことにより前のページにもどってプログラムを確認したり，修正したりすることができる.

また，CONTROL+CのコードはSTOPキーのコードと同じなので，STOPキーを押しても同じようになる.

以上のように，カーソルの移動に関しては，最近のワード•マスターとワード•スターは，ほとんど同じになっている.

しかし，古いバージョンのものでは，かなり違っているので，そのような場合はWM.HLPというヘルプ•ファイルを参考にすればよい.

3.1.3　テキスト修正の方法

テキストの修正作業を分類してみると，次に示すような3種類の作業の組み合わせであることがわかる.

(1)　打ち直し……以前に入力したものの上から正しいテキストで打ち直すこと.
(2)　挿　　入……以前に入力した部分の不足部分を間に入れてやること.
(3)　削　　除……不要部分を取り去ること.

また，これら3種類の作業も行単位の場合と文字単位の場合とが考えられる.

比較的簡単にできる行単位の修正作業から説明していくことになるが，文字単位の修正作業の説明に入る前に，挿入と削除に関するコントロールをまとめておくと，次のようになる.

CONTROL－O	インサート・モード(挿入モード)と重ねがきモードの切り替え.
DEL	カーソルの左側の文字を1文字削除する.
CONTROL－G	カーソル位置の1文字を削除する.
CONTROL－U	カーソルより左側の文字列を削除する.
CONTROL－K	カーソルより右側の文字列を削除する.
CONTROL－Y	カーソルのある行全体を削除する.
CONTROL－N	カーソル位置に改行を挿入する.
CONTROL－¥	カーソルの左側の単語を削除する.
CONTROL－T	カーソルの右側の単語を削除する.

(A) 行単位の修正作業

(1) 行単位の打ち直し

ワード・マスターなどのスクリーン・エディタの場合には，部分的に修正して1行を完成するのが本来の使い方で，行単位の打ち直しはあまり現実的とはいえない．しかし一応，操作方法を説明することとする．

```
MSG:                        ; メッセージ ヲ ヒョウジスル.
LD      (HL),A      ; 1 モジ トリダス. ←──位置がずれている.
        AND     A           ; シュウリョウ コード カ シラベル.
        RET     Z           ; シュウリョウ コード ナラ RETURN
        CALL    DISP        ; メッセージ ナラ ヒョウジスル.
        JMP     MSG         ; ツギノ モジ ノ テスト ヘ.
```

＜修正前のテキスト＞

例えば，上のリストで位置がずれている行を正しい位置になるように打ち直してみることにす

る．まず，カーソルを位置ずれを起こしている行の先頭に合わせる．次に正しい位置に入力した文字がいくように，1行入力する．

```
LD  (HL),A         ；1 モジ トリダス.      ←─元の行の先頭にカーソルをセット
         LD    A,(HL)     ；1 モジ トリダス. ←─正しい位置に入力する
```

すると修正した結果は，次のようになる．

```
MSG；                    ；メッセージ ヲ ヒョウジスル.
         LD    (HL),A    ；1 モジ トリダス.       ←        位置 OK
         AND   A         ；シュウリョウ コード カ シラベル.
         RET   Z         ；シュウリョウ コード ナラ RETURN
         CALL  DISP      ；メッセージ ナラ ヒョウジスル.
         JMP   MSG       ；ツギノ モジ ノ テスト ヘ.
```

＜修正後のテキスト＞

特にここで注意する必要があるのは，スペース•キーとタブ•キーの使い分けであろう．この2つのキーは，見た目には同じように見えるがエディタの操作上は全く違う．タブ•キーは，通常8個分以下のスペースを画面上で占めるが実際には1文字（タブコード）の扱いである．これに対してスペース•キーは，画面上に表示されている文字数分の扱いとなる．

このことは，画面上でカーソルを移動してみるとタブ•キーの所ではカーソルが何文字分も進むことによって確認できるので十分観察されたい．

```
MSG；                    ；メッセージ ヲ ヒョウジスル.
         LD    (HL),A     ；1 モジ トリダス. ←─カーソル移動(CONTROL-D)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
MSG；                    ；メッセージ ヲ ヒョウジスル.
         LD    (HL),A     ；1 モジ トリダス. ←─さらに，カーソル移動
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
MSG：                    ；メッセージ ヲ ヒョウジスル.
         LD   (HL),A      ；1 モジ トリダス. ←─タブ位置でカーソルがとぶ
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
MSG：                    ；メッセージ ヲ ヒョウジスル.
         LD    (HL),A     ；1 モジ トリダス. ←─さらに，カーソル移動
```

(2) 行単位の挿入

行単位の挿入は，プログラム中に命令を何ステップか追加する場合に大変よく行なう操作方法である．行単位の挿入を行なうのにまず大切なのは，カーソルをどこにセットするかということである．このカーソルを，1行挿入したときに後方にずれていく行の先頭にセットする．少々わかりにくいかもしれないが，要は1行とは改行位置までをいうのであるから，挿入位置は次の行になる予定の行の先頭となるのである．カーソルを正しく挿入位置にセットしたら，後はインサート・モード（トグル式のCONTROL+Oを入力して）にし，挿入する行を入力し改行する．複数行ある場合はすべての行の挿入が終了したらインサート・モードから抜け出し（CONTROL+O を入力して）挿入の終了となる．

例えば，次のリストには“CALL DISP”と“JMP MSG”の間に“INC HL”という命令が抜けており正しい動作をしない．そこで“INC HL”を挿入するためにカーソルを挿入後，次の行となる“JMP MSG”の先頭の文字である「J」に合わせCONTROL+Oを入力しインサート・モードにする．このときインサート・モードになった印にカーソル位置の文字が「>」になったり，画面中にINSERT ONの表示が現われる．次に挿入する“INC HL”を入力し改行する．行の挿入は，これで終了なのでCONTROL+Oを入力してインサート・モードから抜け出して挿入の終了となる．**ただし，ワード・スターの場合はCONTROL+Oの変わりに，CONTROL+Vになる．**

```
MSG;                  ; メッセージ ヲ ヒョウジスル.
        LD    (HL),A  ; 1 モジ トリダス.
        AND   A       ; シュウリョウ コード カ シラベル.
        RET   Z       ; シュウリョウ コード ナラ RETURN
        CALL  DISP    ; メッセージ ナラ ヒョウジスル.
        JMP   MSG     ; ツギノ モジ ノ テスト ヘ. ←──次行にカーソルをセット
-----------------------------------------------------------------------
MSG:                  ; メッセージ ヲ ヒョウジスル.
        LD    (HL),A  ; 1 モジ トリダス.
        AND   A       ; シュウリョウ コード カ シラベル.
        RET   Z       ; シュウリョウ コード ナラ RETURN
        CALL  DISP    ; メッセージ ナラ ヒョウジスル.
        >MP   MSG     ; ツギノ モジ ノ テスト ヘ. ←── CONTROL-Oで挿入モ
                                                        ードに
-----------------------------------------------------------------------
MSG:                  ; メッセージ ヲ ヒョウジスル.
        LD    (HL),A  ; 1 モジ トリダス.
        AND   A       ; シュウリョウ コード カ シラベル.
```

```
        RET     Z           ; シュウリョウ コード ナラ RETURN
        CALL    DISP        ; メッセージ ナラ ヒョウジスル.
        INC     HL▒MP  MSG      ; ツギノ モジ ノ テスト ヘ.←─行を入力し改行
--------------------------------------------------------------------
MSG:                        ; メッセージ ヲ ヒョウジスル.
        LD      (HL),A      ; 1 モジ トリダス.
        AND     A           ; シュウリョウ コード カ シラベル.
        RET     Z           ; シュウリョウ コード ナラ RETURN
        CALL    DISP        ; メッセージ ナラ ヒョウジスル.
        INC     HL          ←新しく挿入された行
        ▒JMP    MSG         ; ツギノ モジ ノ テスト ヘ.← CONTROL-Oで挿入モー
                                                         ドから抜け出す(完成)
```

(3) 行単位の削除

行単位の削除は，CONTROL-Yを使って行なう．まず，削除する行にカーソルをセットし，CONTROL-Yを入力すると1行削除される．必要なら，続けてCONTROL-Yを入力してもよい．

```
MSG:                        ; メッセージ ヲ ヒョウジスル.
        ▒LD     (HL),A      ; 1 モジ トリダス. ←─削除する行にカーソルをセット
        AND     A           ; シュウリョウ コード カ シラベル.
        RET     Z           ; シュウリョウ コード ナラ RETURN
        CALL    DISP        ; メッセージ ナラ ヒョウジスル.
        INC     HL          ; ツギノ モジ ニ ポインタ ヲ ススメル.
        JMP     MSG         ; ツギノ モジ ノ テスト ヘ.
--------------------------------------------------------------------
MSG:                        ; メッセージ ヲ ヒョウジスル.
        ▒AND    A           ; シュウリョウ コード カ シラベル.←─CONTROL-Yで1
                                                         行削除
        RET     Z           ; シュウリョウ コード ナラ RETURN
        CALL    DISP        ; メッセージ ナラ ヒョウジスル.
        INC     HL          ; ツギノ モジ ニ ポインタ ヲ ススメル.
        JMP     .MSG        ; ツギノ モジ ノ テスト ヘ.
--------------------------------------------------------------------
MSG:                        ; メッセージ ヲ ヒョウジスル.
```

```
        CALL    DISP        ; メッセージ ナラ ヒョウジスル.←─さらに2回CONTROL-
                                                            Yを押す
        INC     HL          ; ツギノ モジ ニ ポインタ ヲ ススメル.
        JMP     MSG         ; ツギノ モジ ノ テスト ヘ.
```

(B)　文字単位の修正作業

(1)　1文字打ち直し

すでに入力したプログラム中の1文字打ち直すには，まずカーソルを打ち直す文字の上にセットし，その上から，正しい文字を入力してやればよい．ただし，このときにインサート・モードになっているとカーソルの位置にあった文字以降が右側に1文字分ずれ，カーソルのある所に新たに入力した文字が挿入されてしまう．したがって，インサート・モードに，なっていないことを確認してから文字を入力するようにしなければならない．

```
MSG:                        ; メッセージ ヲ ヒョウジスル.
        LD      (HL), A     ; 1 モジ トリダス.←─打ち直す文字にカーソルをセット
..........................................................................
MSG:                        ; メッセージ ヲ ヒョウジスル.
        LD      (HL), A     ; 1 モジ トリダス.←─正しい文字を入力する
```

(2)　1文字挿入

すでに入力したプログラム中に1文字挿入するには，挿入する位置にカーソルをセットして，次に挿入する文字を入力するのであるが，そのためにはインサート・モードになっていなければならない．もしインサート・モードになっていなければ，CONTROL+Oでインサート・モードにしてから挿入する文字を入力する．

```
MSG:                        ; メッセージ ヲ ヒョウジスル.
        LD      (), A       ; 1 モジ トリダス. ←─挿入する位置にカーソルをセット
..........................................................................
MSG:                        ; メッセージ ヲ ヒョウジスル.
        LD      (H), A      ; 1 モジ トリダス. ←─Hを入力
..........................................................................
MSG:                        ; メッセージ ヲ ヒョウジスル.
        LD      (HL), A     ; 1 モジ トリダス. ←─Lを入力
```

(3) 1文字削除

すでに入力したプログラム中の文字を削除するには，まずカーソルを削除する文字の上にセットする．次に，CONTROL+Gを入力すると，カーソル位置の文字が1文字削除される．

```
MSG:                    ; メッセージ ヲ ヒョウジスル.
        LD      (HL), A ; 1 モジ トリダス.←—カーソルを削除する文字の上に移動
-------------------------------------------------------------------------
MSG:                    ; メッセージ ヲ ヒョウジスル.
        LD      HL), A  ; 1 モジ トリダス. ←—CONTROL+Gを入力
-------------------------------------------------------------------------
MSG:                    ; メッセージ ヲ ヒョウジスル.
        LD      L), A   ; 1 モジ トリダス. ←—さらに，CONTROL+Gを入力
```

以上のような修正方法をうまく組み合わせて，プログラムの修正を行なうようにする．慣れないうちは，多少大変であるが，キャラクタ•エディタを使うよりは，視覚的にはるかに楽に作業が進められるのがワード•マスターやワード•スター等のスクリーン•エディタのよいところといえる．

3.1.4 文字列の検索と置換

作成するプログラムが1画面から3画面程度の比較的短いものなら，何行目あたりにどのようなルーチンがあったかだいたい見当がつくのであまり問題になることはないが，本格的なプログラムとなるとプログラムの行数も2000行を軽く越してしまうことが多い．このように画面数で100画面を越してしまう場合などは，どこにどのようなルーチンがあったか，簡単にわかるものではない．このようなときに強力な武器になってくれるのが，“文字列の検索機能”である．

また，このような長いプログラムにおいて何回も使われているような単語を一度に他の単語に変換するようなときに非常に便利な機能が“文字列の置換機能”である． 実際に文字列の検索と置換の説明に入る前に，どのようなコマンドがあるのかをまとめてみると，次の表のようになる．

コマンド	機　能
+−nF キー $	キーで短い検索をする．
+−nN キー $	キーで長い検索をする．
+−nS キー $テキスト $	キーで短い検索をし，見つかれば新しいテキストで置換する．

+−nRキー$テキスト$	キーで長い検索をし，見つかれば新しいテキストで置換する．

注）Fは FIND，Nは NEXT，Sは SUBSTITUTE，Rは REPLACEの略称で検索と置換の意味で，またnはコマンドを繰り返す回数を示している．

この表の中に出てくる短い検索と，長い検索の違いはカーソル位置から2000文字以内，もしくはメモリ内に格納できるテキストの範囲である．したがって，2000文字以内で，かつメモリ内なら短い検索で，2000文字を越えたりメモリ中に存在しない場合は長い検索ということになる．

（A）文字列の検索

文字列の検索には短い検索の+−nFキー$と，長い検索の+−nNキー$の2つがある．これら2つのコマンドの違いは，検索する範囲が長いか短いかだけなので，ここでは，説明のしやすい短い検索の+−nFキー$を使って説明することにする．

まず検索をするには，検索するキーを選定することが必要となる．検索するキーは，なるべく短くして他の部分とハッキリと識別できるものがよい．ただしここで識別できるということは，コンピュータにとって識別できるということで，いくらわれわれ人間に識別可能なキーでもコンピュータに識別できなければだめである．もし，そのようなキーをまちがって選定した場合には，他の関係ない部分まで検索してしまうこととなる．次にコマンド・モードに移りカーソルを検索開始位置にセットし，実際に検索するということになる．

MSG: 　　MOV　A,(M) 　　ANA　A 　　RZ *_	・ESCキーを押してコマンド・モードに入る． コマンド・モードに入ると*が画面下部に現われ，カーソルが点滅しコマンド待ちになる．
MSG: 　　MOV　A,(M) 　　ANA　A 　　RZ *B	・Bコマンドでカーソルを検索開始位置である先頭の行にセット．
MSG: 　　MOV　A,(M) 　　ANA　A	・検索する行範囲と文字列をセットして検索に入る． ここでは，カーソル位置から3行の範囲内にあるANとい

RZ *3FAN$	う文字列をさがすということになる. ($はESCまたは改行を入力することを意味する.)
MSG: MOV A,(M) ANA A RZ *V	・ビデオ・モードにもどるためにVコマンドを実行するとカーソルが検索した文字列の後ろの文字位置で停止しているのが発見できる.

文字列の検索は，修正位置や参照位置の発見に非常に有効な手段となる．この機能を上手に使えばいちいち長いプログラムを打ち出さなくとも，見たいときに見たい所を自由に画面に表示させることができる.

(B) 文字列の置換

文字列の置換にも，短い検索の後に置換を行なう +-nS キー $ テキスト $ と長い検索の後に置換を行なう +-nR キー $ テキスト $ の 2 つがある．ここでは，説明のしやすい短い検索の後に置換を行なう +-nS キー $ テキスト$ を使って説明することにする.

文字列の置換では，まず置換を行なう前に検索を行なう．したがって文字列の検索のときと同様に，なるべく短くて他の部分とまちがわれないように検索する文字列を選定することが必要不可欠となる．特にこの置換の操作では，キーで指定された文字列を無条件で指定されたテキストの文字列に書き換えてしまうので十分注意して行なうようにしなければならない．万一，関係のない部分を書き換えてしまったような場合には，単純に逆の置換を行なって置換する前の状態にもどすというわけにはいかないので，大変なことになる.

MSG: MOV A,(M) ANA A RZ *_	・ESCキーを押してコマンド・モードに入る. コマンド・モードに入ると*が画面下部に現われ，カーソルが点滅しコマンド待ちになる.
MSG: MOV A,(M) ANA A RZ *B	・Bコマンドでカーソルを検索開始位置である先頭の行にセット.

MSG: MOV　A,（M） ANA　A RZ *3SMOVLD	・検索する行範囲と検索するキー，および置換する文字列をセットして置換に入る． ここでは，カーソル位置から3行の範囲内にあるMOVを探し，"LD"の文字列で置き換えることになる． （$はESCまたは改行を入力することを意味する．）
MSG: LD　A,（M） ANA　A RZ *V	・ビデオ・モードにもどるためにVコマンドを実行するとカーソルが置換した文字列の後ろの文字位置で停止しているのが発見できる．

練習のために，

キー	テキスト
（M）	（HL）
AND	AND
RZ	RET　Z

の3種類の置換を同様にして実行してみよう．最終結果は，次のようになる．

```
MSG:
        LD      A,(HL)
        AND     A
        RET     Z
```

すなわち，置換する前は8080形式のアセンブリ表記であったが，置換後はZ80形式のアセンブリ表記に変わっている．

説明の都合上，検索のときも置換のときも1ステップごと説明をしたのであるが，実際にこれらの操作を行なうには，次のようにまとめて行なうことができる．

MSG: MOV　A,（M） ANA　A	・ESCキーを押してコマンド・モードに入る． コマンド・モードに入ると*が画面下部に現われ，カーソ

RZ *_	ルが点滅しコマンド待ちになる.
MSG: 　　　LD　　A, (M) 　　　ANA　 A 　　　RZ *B3SMOV$LD $V	・Bコマンドでカーソルを検索開始位置であるファイルの先頭の行にセット. ・検索する行範囲と検索するキー，および置換する文字列をセットして置換に入る. ・ビデオ・モードにもどるためにVコマンドを実行するとカーソルが置換した文字列の後ろの文字位置で停止しているのが発見できる.

ところで，この例では検索する範囲をカーソル位置から3行としているが，バッファ内のテキストの最後尾まで検索するようにもできる．そのためには，長い検索の後に置換を行なう+−nRキー$テキスト$を使う．また，置換する回数が今の場合だと1回であるが何回か続けて置換を行なう場合には，置換回数を指定して行なえばよい.

例えば，3回繰り返すなら

*B3<SMOV$LD $>V@ ……@は，改行の意味で，<>は繰り返す範囲を表わす

のようにコマンドを入力すればよい．何回繰り返せばよいかわからないが，バッファ内のテキストの最後尾まで，該当するものはすべて置換したいようなときには，

*B<SMOV$LD $>V@……繰り返す回数が省略されている

のようにコマンドを入力する．このように，繰り返す回数を省略すると65535回置換を繰り返し行なうことになる．したがって，長い検索の後に置換を行なう+−nRキー$テキスト$ を使って

*B<RMOV$LD $>V@……+−nRキー$テキスト$を使う

のようにコマンドを入力すれば，ファイルの最初から最後までの該当するキーがテキストによってすべて置換されることになる.

ところで，上記のコマンドを次のように変えた場合には，どのようなことになるのであろうか.

*B<RMOV$LD $V>@……Vを<>の中に入れる

このようにすると，MOVを探して“LD”に置換した後，ビデオ・モードに移りカーソルが置換した文字列の後ろで点滅しながら表示が停止する．しかし，ESCキーを押すと次のMOVの置換に移る．この操作をMOVが発見できなくなるまで，繰り返し行なうことができる．ただ，残念なことに，この操作は途中で打ち切ることができない．コマンドを入力して改行した後でまちがいに気がついても，後の祭ということになるので使用にあたっては十分注意することである.

(C)　ワード・スターの場合の検索と置換

ワード・スターでの文字列の検索と置換は，CONTROLE−Qによって表示されるQUICK MENUのなかにあるFとAのコマンドによって行なわれる．またファイルの先頭にポインタを移動

するのも，同じQUICK MENUのなかにあるＲコマンドである．したがってワード・スターでは，CONTROLE−QとR，F，Aの３つのコマンドを組み合わせて使い検索と置換を行なうことになる．また，ワード・スターやワード・マスターではコントロール・キーを“^”で表わしているので，ここでも“^”を使って表わすことにする．

参考のために，QUICK MENUを次に示しておく．

```
<<<    QUICK   MENU      >>>
---Cursor Movement---           | -Delete-    | --Miscellaneous--        | --Other Me
S  left side   D  right side    | Y line rt   | F Find text in file      |  (from Main
E  top scren   X  bottom scrn   | DEL lin lf  | A Find & Replace         | ^J  Help  ^K
R  top file    C  end file      |             | L Find Misspelling       | ^Q Quick  ^P
B  top block   K  end block     |             | Q repeat command or      | ^O Onscreen
0-9  marker Z  down W  up       |             |   key untile space       | Space Bar retu
P  Previous    V  last Find or Block          |   bar or other key       | you to main Me
```

（１） **検索の方法**

では，実際に検索を行なってみることにする．

1・・・^QRと入力しカーソルをファイルの先頭にセットする．

2・・・^QFと入力し検索のモードに入る．すると，次のようなメニューが画面に現われる．

```
^QF    B:WORD.TST    PAGE 1 LINE 1 COL 01          INSERT ON
  ^S=delete character          ^Y=delete entry    ^F=find directory
  ^D=restore character         ^R=Restore entry   ^U=cancel command
      FIND? LD OPTIONS? (? FOR INFO)
```

3・・・ FIND?に続けて検索する文字列”LD”を入力し改行する．

4・・・ 次に，OPTION?(FOR INFO)と表示がでて検索するためのオプションを問い合わせてくるが，ここでは単に改行を入力する．

5・・・ すると，検索した ”LD”の次の文字にカーソルが移動する．

```
MSG:
         LD     A,(HL)
         AND    A
         RET    Z
```

6 ・・・ このとき，^L と入力すれば次の "LD" を検索しに行く．発見できなければ NOT FOUND の後に，Press ESCAPE KEY の表示が表われワード・スターにもどることになる．

ワード・スターはワード・マスターと違い非常にコマンドが多いので1つのコマンドを実行するのに今回使用した，^QF のように2文字入力する場合があるので少し大変かもしれない．しかし，ある程度使い慣れてくると，画面の表示を待たずに先行入力していっても正しく動作するので便利さのほうが目だってくるに違いない．

(2) 置換の方法

では，実際に置換を行なってみることにする．

1 ・・・ ^QR と入力し，カーソルをファイルの先頭にセットする．

2 ・・・ ^QA と入力し，置換のモードに入る．すると，次のようなメニューが画面に現われる．

```
^QF   B:WORD.TST   PAGE 1 LINE 1 COL 01          INSERT ON
  ^S=delete character         ^Y=delete entry     ^F=find directory
  ^D=restore character        ^R=Restore entry    ^U=cancel command
     FIND? MOV REPLACE WITH? LD
     OPTIONS? (? FOR INFO)
```

3 ・・・ FIND? に続けて置換する文字列 "MOV" を入力し，改行する．

4 ・・・ すると，REPLACE WITH? のように何に置換するか聞いてくるので "LD" と入力して改行をする．

5 ・・・ 次に，OPTION?(FOR INFO) と表示が出て検索するためのオプションを問い合わせてくるが，ここでは単に改行を入力する．

6 ・・・ すると "MOV" の所でカーソルが停止し，置換操作を実行するかどうか (REPLACE Y/N) を聞いてくるので，置換するなら Y を，置換しないなら N を入力する．ここでは，Y と入力する．

7 ・・・ すると置換を行ない，置換した "LD" の次にカーソルが移動する．

```
MSG:
          LD    A,(M)
          ANA   A
          RZ
```

8・・・ このとき，^Lと入力すれば次の"MOV"を置換しに行くが，発見できなければNOT FOUNDの後にPress ESCAPE KEYが現われ，ワード・スターにもどることになる．

5でOPTION？（FOR INFO）と表示が出てきたときにNと入力すると，ワード•マスターの場合と同じく該当する文字列を無条件に置換してしまう．

3.1.5 テキストのRead・Write

ここでは，エディタ操作法の最後として，エディタで作成したファイルをディスクに登録するいくつかの方法について述べることにする．

(A) ファイルを更新して作業を終了する場合

エディタを使ってプログラムの入力がすべて完了したら，当然その結果をディスクに登録することになる．また，前回入力しておいたプログラムのまちがいを修正し，修正した新しいファイルをディスクに登録するような場合，ファイルを更新して作業の結果をディスクに登録する．このように結果をディスクに登録したりファイルを更新して作業を終了するような場合には，Eコマンド（EXIT）を実行する．

MSG: 　　LD　A,(HL) 　　AND　A 　　RET　Z *_	・ESCAPEキーを入力して，コマンド・モード入る．
MSG: 　　LD　A,(HL) 　　AND　A 　　RET　Z *E	・Eコマンドを実行して，作業を終了する． （このときに，作成または更新されるファイルは，最初ワード・マスターを起動したときに指定したファイル名のファイルで，途中での変更はできない．）

(B) ファイルを更新しないで作業を終了する場合

ファイルの内容を確認するためや，置換操作のミスで修正困難となり作業を取り消したい場合などに，Qコマンド（QUIT）を使用するとよい．Qコマンドを実行すると，それまでの作業を無効とし，元のファイルに影響を与えずに作業を終了することができる．

MSG: 　　LD　A,(HL) 　　AND　A 　　RET　Z *Q ABORT(Y/N)?Y	・Qコマンドを実行する. ・すると確認のためABORT(Y/N)?と聞いてくる.もしQコマンドの実行を取り消したければ,Nを入力すれば,Qコマンドを実行する前の状態にもどることができる. ・Yと入力するとエディタを抜けてCP/Mにもどる.

(C)　元のファイルにもどって修正作業を再開したい場合

修正作業中に致命的なまちがいに気がついて初めから作業をやり直したいような場合には,それまでの修正を無効にして(Qコマンド),再びワード•マスターを起動すればよい.しかしこのような場合には,Oコマンドを使えば,再度,開始時に指定したファイルを自動的にロードしてくれるので,再びワード•マスターを起動する手間が省けることになる.

MSG: 　　LD　A,(HL) 　　AND　A 　　RET　Z *O ABORT(Y/N)?Y	・Oコマンドを実行する. ・すると確認のためABORT(Y/N)?と聞いてくる.もしOコマンドの実行を取り消したければ,Nを入力すれば,Oコマンドを実行する前の状態にもどることができる. ・Yと入力すると元のファイルをロードして修正に入る.

(D)　修正中の中間結果をディスクに登録したい場合

最近ではめったにないことであるが,突然のトラブルによって修正中のファイルが破壊してしまうことがある.このような事態を避けるために修正中の中間結果をディスクに登録しておけばよい.そうすれば,万一ファイルが破損するようなことが起きてもBAKファイルを採用することにより,少なくとも中間結果までは復旧することが可能になる.ただしこのHコマンドを実行するとディスク中のBAKファイルが更新されるので,前述のOコマンドで元のファイルにもどった場合には,最後にHコマンドを実行したときの状態までが限度となる.

MSG: 　　LD　A,(HL) 　　ANDA 　　RET　Z *H	・まちがって修正してしまった場合,Hコマンドを実行する.

MSG: 　　LD　A,(HL) 　　AND　A 　　RET　Z	・Hコマンドを実行した結果，ファイルの内容が元のファイルの内容にもどっていることが発見できる.

以上の4種類のコマンドがワード・マスターには準備されている.

ただし，一般的にはEコマンドとQコマンドはよく使われるが，HコマンドやOコマンドを使うことは非常に稀である.

3.2　マクロ・アセンブラ(MACRO−80)の効果的な使用法

この節では，リロケータブル・マクロアセンブラとして，マイクロソフト社のMACRO−80の使用法を，次の手順で解説する．

3.2.1　ソース・ファイルの形式

(A)　ソース・ファイルの名称

CP/Mのファイル名として使用できるものは，この“MACRO−80”のソース・ファイル名として使用できるが，ファイル名拡張子(ファイル名としてピリオドの後に付けられる3文字の識別子)としては“.MAC”を指定するのがよい．これは，MACRO−80の省略時のファイル名拡張子であり，このような拡張子を指定しておくことによって，以後，ファイル名の指定を行なうときに，この拡張子を省略することができる．

ただし，ライブラリ・モジュールのソース・ファイルのようなものを作成する場合には拡張子として，“.LIB”のような名前を付けておくほうが管理しやすい場合もあるので目的に合わせたネーミングをしておくのがよいであろう．

注)　ファイル名中に，“$，&”のような記号を含むものは，MACRO−80のソース・ファイル名としては，使用することはできない．すなわち，ソース・ファイル名として使用できるのは，“アルファベットと数字の組み合わせ”ということになる．

(B)　シンボル(名前)に関する規則

MACRO−80で使用できるシンボルは，任意の長さのものが使用できるが，リンクローダが区別をできるのは，

- ラベルは最初の16文字．
- 共通参照記号および外部参照記号は，最初の6文字のみである．

となる．また，このシンボルとして使用できる文字は，

A～Z	0～9	$	.	?	@	_

であるが，1文字目は数字または，アンダーバー(_)であってはならない($,.,?,@の記号は許される)．また，アルファベットの小文字は大文字に変換される．

注） ピリオド“.”ではじまるシンボルは，マクロ定義のローカル変数として使用されるので，ユーザープログラムの中での使用はさしひかえたほうがよい．

(C) 数値の表記法

“.RADIX擬似命令”により基数の指定が行なわれていない限り，数値は，10進数として扱われる．その他の数値を表わすには，次のようになる．

2進数	末尾にBを付ける．	例． 10110001B
8進数	末尾にOまたは，Qを付ける．	例． 732O，23Q
16進数	末尾にHを付ける．	例． 45DFH，2CH
	先頭の文字がアルファベットの場合には，先頭に0を付ける．	
		例． 0A03FH，0A3H

(D) 1行のフォーマット

MACRO－80のステートメント行は，次のようなフォーマットでなければならない．

シンボル フィールド	オペレーション フィールド	式 フィールド	コメント フィールド

シンボルフィールド	このフィールドにはシンボル(名前)を記述する． EQU擬似命令，マクロ定義擬似命令以外のときは，最後にコロン(:)を付ける．
オペレーションフィールド	このフィールドには，オペコード，擬似命令，マクロ名を記述する．
式フィールド	このフィールドには，オペレーションフィールドに対するオペランドを記述する．
コメントフィールド	このフィールドの先頭にはセミコロン(;)を置き，その後にコメントを記述する．

注） それぞれのフィールドは，省略することが可能な場合がある．

(E) 基本的なソース・プログラムの形式

MACRO−80のソース・プログラムの基本的なフォーマットは，次の2種類の形になる．

```
1) アブソリュート型のプログラムにする場合
  ASEG ←──────────── アブソリュート型指定
  ORG    ****H ←──────── スタートアドレス

  ┌ - - - - - - - - - - - ┐
  : プログラム・エリア    :
  :       and             :
  : データ・エリア        :
  └ - - - - - - - - - - - ┘

  END
2) リロケータブル型のプログラムにする場合
  CSEG ←──────────── プログラムコード・エリア指定
START:: ←──────────── プログラムスタート位置
                       共通参照記号にしておく
  ┌ - - - - - - - - - - - ┐
  : プログラム・エリア    :
  └ - - - - - - - - - - - ┘

  DSEG ←──────────── データ・エリア指定

  ┌ - - - - - - - - - - - ┐
  : データ・エリア        :
  └ - - - - - - - - - - - ┘

  END    START ←──── リンクローダへの情報
```

3.2.2 使用できる演算子とその機能

演算子には，算術演算を行なう“算術演算子”，論理演算を行なう“論理演算子”，2数間の大小関係を比較する“関係演算子”，特殊な機能をもつ“特殊演算子”がある．これらを優先順位の高い順番に並べると，次のような表になる．

LOW, HIGH
*, /, MOD, SHR, SHL
−(単項マイナス)
+, −

```
EQ, NE, LT, LE, GT, GE
NOT
AND
OR, XOR
```

これらの優先順位は，括弧を使用することにより変えられる．また，+，−，*，/ 以外の演算子は，1つ以上のスペースによりオペランドと区別されていなければならない．

(A) 算術演算子

算術演算子には，次の10個のものがあり，一般の算術機能(加算，減算，乗算，除算，剰余)と2つのシフト演算子，2つのバイト分離演算子をもっている．"+" および "−" を除いてオペランドは定数でなければならない．算術演算子の種類とその機能は，次のようになる．

*	乗算
/	除算
MOD	剰余：左オペランドを右オペランドで割って，その余りを返す．両方のオペランドとも絶対値でなければならない．
SHR	右シフト：SHRの後には，オペランドを右シフトさせる数値が入る．
SHL	左シフト：SHLの後には，オペランドを左シフトさせる数値が入る．
HIGH	アドレス式の上位8ビットを分離する．
LOW	アドレス式の下位8ビットを分離する．
−	単項マイナス：それ以降に続く数値が負であることを示す．
+	加算：1方のオペランドは定数でなければならない．
−	減算：両方のオペランドとも非定数でもよい．

アセンブル例

```
0100'   4400   DW   2F44H*100H
0102'   002F   DW   2F44H/100H
0104'   0044   DW   2F44H MOD 100H
0106'   0F     DB   11110000B SHR  4
0107'   F0     DB   00001111B SHL  4
0108'   2F     DB   HIGH 2F44H
0109'   44     DB   LOW 2F44H
010A'   FF     DB   -1              ；単項マイナス
010B'   06     DB   5+1
010C'   04     DB   5-1
```

(B)　論理演算子

論理演算子は，各オペランドの対応するビット位置のバイナリ値に対する論理演算を行ない，その結果を返す．論理演算には，次のものがある．

	NOT	論理 NOT	
	AND	論理 AND	
	OR	論理 OR	
	XOR	排他的論理和	
	アセンブル例		
0100'	0F	DB	NOT 11110000B
0101'	B0	DB	10110101B AND 11110000B
0102'	F5	DB	10110101B OR 11110000B
0103'	45	DB	10110101B XOR 11110000B

注）これらのオペランドは，定数でなければならない．

(C)　関係演算子

関係演算子は，2つの定数オペランドの比較を行なう．その比較の結果，真(FFH)または，偽(00H)が返される．これらの関係演算子は，条件付き擬似命令，および条件判定命令とともに使用されることが多く，プログラムの流れを制御することに用いられる．

関係演算子には，次のものがある．

	EQ	等しい	
	NE	等しくない	
	LT	より小さい	
	LE	小さいか等しい	
	GT	より大きい	
	GE	大きいか等しい	
	アセンブル例		
0100'	FF	DB	100 EQ 100
0101'	00	DB	100 NE 100
0102'	00	DB	100 LT 50
0103'	FF	DB	100 LE 100
0104'	FF	DB	100 GT 50
0105'	FF	DB	100 GE 100

(D) 特殊演算子

これらの演算子は単独で使用することは少なく，通常はマクロ定義の内部で使用するものが多い．これらの演算子には，次の2つのものがある．

NUL	パラメータがヌル(空白)の場合真を返す．この演算子の後ろの部分は，すべてこの演算子のパラメータとしてみなされる．
TYPE	パラメータにはシンボルをとり，そのシンボルが外部参照記号かということと，そのシンボルがどのセグメントに配置されているかを返す．

◎　TYPE 演算子で得られる情報

ビット7 …… 外部参照ビット	1 ：外部参照を含む	0 ：内部参照
ビット5 …… 定義ビット	1 ：定義済	0 ：未定義
ビット1，0 … セグメント	00： ASEG	01： CSEG
	10： DSEG	11： COMMON

3.2.3　擬似命令の種類とその機能

アセンブリ言語の機能を評価する場合，そのアセンブリ言語に用意されている擬似命令を調べれば，おおよその検討がつく．CP/M にバンドリングされているアブソリュート型のアセンブラである“ASM”などの場合には，この擬似命令の数が非常に少ない．

それに比較して，この MACRO-80 の場合には，データ関係やリスト制御関係の擬似命令だけでなく，リロケータブル(再配置)関係の擬似命令や，マクロ定義に関係する擬似命令というこれまでにはなかった種類の擬似命令が用意されている．それだけこのアセンブラの機能が大きいということが想像できるであろう．ここでは，これらを機能別に分けて解説することにする．
([ ]記号はオプションを表わす)

(A) データ定義関係擬似命令

これらは ASM でもお馴染みのものであるが，Z80のニーモニックの関係により，いくつかの異なる名称を使用することができる．これらをまとめると，次のようになる．

DB (DEFB/DEFM)	式または文字列	バイト定義
DW(DEFW)	式	ワード定義
DS (DEFS)	式	領域の定義
DC	ストリング	文字の定義

EQU	式	値のセット
SET(ASET/DEFL)	式	同上(ただし，再定義可能)

注) ・式とは，数値，ラベルなどと演算子を組み合わせたものである．
・Z80ニーモニックを使用する場合には，SETの代わりにASETを使用する．

アセンブル例

0000' 1F 22 2D	DATA1:	DB	1FH, 22H, 2DH	; 数値
0003' 54 65 73 74	MSG:	DC	'Test'	; 文字列
0007' 8000	ADDR1:	DW	8000H	
0009'	KBUF:	DS	128	
0014	Z	EQU	20	
000A	Z	ASET	10	; 再定義

(B) リスト制御関係擬似命令

このリスト関係の擬似命令には，リスト出力に関するものと，コメントに関するものがある．これらの擬似命令は，上手に使用することによってドキュメンテーションの整備はもとより，デバッグの効率を向上させることもできる．これらをまとめると，次のようになる．

PAGE/$EJECT [式]

◇リスト出力に対して改ページを行なう．
◇パラメータとして式が指定された場合には，その式の値が1ページ当たりの行数となる．この値は10～255でなければならない．1ページの行数の初期値は50行である．

例：PAGE　改ページ
　　PAGE 40　　1ページを40行とする．

TITLE {テキスト}

◇リスト出力の各ページの最初の行に表示するタイトルを指定する．
◇この擬似命令は1つのプログラムに1回のみ使用できる．

例：　TITLE CPM Utility No.1

SUBTTL/$TITLE {テキスト}

◇TITLEの次行に表示するサブタイトルを指定する．
◇テキストは60文字以内にする．
◇この擬似命令は，1つのプログラムにいくつ使用してもよい．ページごとのタイトルと考えればよい．

例： SUBTTL モジュール No.1

.LIST

◇リストを出力する．何も指定しないと，この擬似命令が指定されたのと同じになる．

.XLIST

◇リストの出力を止める．リストを表示したくない部分の前にこの擬似命令を使用する．再び表示を開始する場合には，“.LIST 擬似命令”を使用する．

.CREF

◇クロスリファレンス・リストに関する“.LIST 擬似命令”と同様の機能となる．

.XCREF

◇クロスリファレンス・リストに関する“.XLIST 擬似命令”と同様の機能となる．

.SFCOND

◇偽となる条件ブロックのアセンブル・リストを出力しない．

.LECOND

◇偽となる条件ブロックのアセンブル・リストを出力する．

.TFCOND

◇MACRO−80の/Xスイッチの有無により，偽となる条件ブロックのアセンブル・リストの出力を決定する．

.XALL

◇マクロおよびブロック展開部分のうちオブジェクト・コードを生成する部分のみアセンブル・リストに出力する．

.LALL

◇マクロおよびブロック展開部分のすべてをアセンブル・リストに出力する．

.SALL

◇マクロおよびブロック展開部分をアセンブル・リストに出力しない．

.COMMENT　{区切り記号}{テキスト}{区切り記号}

◇区切り記号から，区切り記号までのテキストをコメントとする．

◇区切り記号は，いずれの文字でもよい．多行にまたがるような長いコメントを記述する場合に使用するとよい．

例：　.COMMENT　*
　　　　　　　　∫　　この間には何行のコメントを書いてもよい
　　　　　　　　*

.PRINTX　{区切り記号}{テキスト}{区切り記号}

◇この擬似命令は，アセンブル時に，テキスト部分が画面に表示される．COMMENT 擬似命令に文法が似ているが，この場合のテキストは，この擬似命令と同一の行になければならない．長いプログラムのアセンブルの過程をチェックするのに使用するとよい．

（C）リロケータブル関係擬似命令

この擬似命令は，リロケータブル・アセンブラ固有の命令で，再配置関係の処理に関する命令である．これらの擬似命令の使用法は，リロケータブル・アセンブラの考え方を理解していないとわかりにくいものが多いので注意を要する．この基本になるものは，そのプログラム内で定義されていて，それ以外のプログラムからも参照される“**共通参照記号**”と，外部のプログラムで共通参照記号として定義されている記号を指定する“**外部参照記号**”と，そのプログラムの内部でのみ使用する“**ローカル記号**”の区別をすることと，リンクローダ(LINK−80)で使用する情報と，ライブラリマネジャ(LIB−80)で必要となる情報を理解することにある．

これらをまとめると，次のようになる．

注）記号とはシンボル(名前)のことである．

ASEG(Absolute SEGment)

◇ORG 擬似命令と組み合わせてアブソリュート(絶対番地)形式のプログラムを作成する．

◇この擬似命令を使用したプログラムは，リロケータブル(再配置可能)なプログラムとはならない．開始アドレスが決定しているようなメイン・プログラムに使用するとよい．

例：　100H 番地から始まるプログラムにする場合

```
ASEG        ; アブソリュート・セグメント設定
ORG 100H    ; 開始番地設定
∫
```

CSEG(Code SEGment)

◇リロケータブルなプログラムで，コード・セグメントを指示する．

◇リロケータブルなプログラムにするためには，この擬似命令と次の“DSEG擬似命令”の間にプログラムを記述する．

DSEG（Data SEGment）

◇リロケータブルなプログラムで，データ・セグメントを指示する．

◇リロケータブルなプログラムにするためには，この擬似命令と“END擬似命令”の間にデータを記述する．

COMMON

◇FORTRANの“COMMONステートメント”との互換性を保つための擬似命令である．

◇複数のモジュール（プログラム）での共通のエリアを定義する．

.PHASE {アドレス}

◇この擬似命令は，リロケータブルなプログラム中で，絶対アドレスによるジャンプやコールを行ないたいときに使用する．すなわち，リロケータブルなプログラムにもかかわらず，このPHASE擬似命令が指定されると，それ以降の部分のジャンプやコールに関するアドレスは，PHASE擬似命令のアドレスからの絶対アドレスとして定義される．しかし，そのオブジェクト・コードは，リロケータブルなコードとなる．

◇この擬似命令は，アセンブル時の状態と，実行時の状態が異なるような場合に有効な命令で，あまり使用することはない．DEPHASE擬似命令が実行されるまで有効である．

例：　◎ PHASE擬似命令を使用した場合

```
0000'                    .PHASE    100H
0100  C3 0104            JP        J1
0103  00                 NOP
0104  00           J1:   NOP
                         .DEPHASE
0005' C3 0000    EXIT:   JP        0
```

◎ PHASEを使用しない場合

```
0000' C3 0004'           JP        J1
0003' 00                 NOP
0004' 00           J1:   NOP
0005' C3 0000    EXIT:   JP        0
```

.DEPHASE

◇ PHASE擬似命令を解除する．

PUBLIC(ENTRY/GLOBAL)　{シンボル}[,{シンボル}...]

◇パラメータとして指定したシンボルを，共通参照記号(シンボル)として定義する．

◇これはまた，シンボルの後にコロン“:”を使用できるものの場合には，二重コロン“::”を使用しても同じ意味になる．

◇ZSIDによるシンボリック・デバッグの場合には，この定義がしてあるシンボルしか使用できない．

例：
```
PUBLIC      STRING,VAL,ASC
STRING::    LD  A,B
```

EXTRN(EXTERNAL/EXT)　{シンボル}[,{シンボル}...]

◇パラメータとして指定したシンボルを，外部参照記号として定義する．

◇これはまた，シンボルの後に，二重イゲタ“##”を使用しても同じ意味になる．

◇この擬似命令により定義されているシンボルは2バイトの値として評価される．

◇この擬似命令により定義されたシンボルは，リンクローダ(LINK-80)でリンクされる他のモジュール内で，共通参照記号として定義されていなければならない．

例：
```
EXTRN   STRING,VAL,ASC
CALL    STRING##
```

BYTE EXTRN　{シンボル}[,{シンボル}...]
BYTE EXTERNAL　{シンボル}[,{シンボル}...]
BYTE EXT　{シンボル}[,{シンボル}...]

◇パラメータとして指定したシンボルを，外部参照記号として定義する．

◇これはまた，シンボルの後に，二重イゲタ“##”を使用しても同じ意味になる．

◇この擬似命令により定義されているシンボルは1バイトの値として評価される．

◇この擬似命令により定義されたシンボルは，リンクローダ(LINK-80)でリンクされる他のモジュール内で，共通参照記号として定義されていなければならない．

例：
```
BYTE EXTRN  DATA1,DATA2
DATA1##  DB   0CH
         LD   A,CR$##
```

◎　リンクローダ(LINK-80)に関するもの

.REQUEST　{ファイル名}[,{ファイル名}...]

◇この擬似命令は，リンクローダ(LINK-80)に対して，このプログラム内で使用している他のモジュールやライブラリプログラムのファイル名を与える．

◇拡張子やドライブ番号を省略した場合には，拡張子は“.REL”，ドライブ番号は現在使用しているドライブ(LOGIN DISK)になる.

◇この擬似命令は，ソースプログラム中のいずれの場所に記述してもまた何回使用してもよい.

◇この擬似命令を使用することによりリンクするときに，リンクローダに対してライブラリプログラムを指示する必要がなくなる（FORTRAN コンパイラなどでは，この擬似命令のファイル名として“FORLIB”が指定されているためにリンク時に自動的に，FORLIB.RELがリンクされるわけである).

例： ライブラリプログラムとして，MYLIB.RELをリンクする場合.

```
.REQUEST  MYLIB
```

◎ ライブラリマネジャ(LIB−80)に関するもの

NAME （'モジュール名'）

◇この擬似命令のパラメータは，ライブラリマネジャ(LIB−80)にて出力されるモジュール名として使用される.

◇このモジュール名は，最初の6文字のみが意味をもつ.

◇この擬似命令も“TITLE擬似命令”と同様に，1つのプログラムに1回のみ使用できる.

◇プログラム中にこの擬似命令がなかった場合には，TITLE擬似命令によるテキストがライブラリマネジャのモジュール名となる．このTITLE擬似命令もない場合には，ソース・ファイルの名前がモジュール名となる.

例： NAME （'STRING'）

注） “('”と“')”は必ず必要である.

(D) 条件付擬似命令

この擬似命令は，アセンブラに対して，アセンブルの条件を与えるもので，マクロ定義を行なう場合に多く用いる．これらの条件付擬似命令は，すべて次のようなフォーマットで使用する.

```
IF** ［パラメータ］
  ～  条件を満たすときの処理
ELSE(オプション)
  ～  条件に反するときの処理
ENDIF
```

これらの条件付擬似命令は任意のレベルの入れ子(nesting)になっていてもよい．しかし，値のエラー(Value error)をさけるために，どの変数もパス1で既知になっていなければならない．また，式をパラメータとしてもつものは，その式の値は絶対値でなければならない．これらの設定があいまいであると，パス1とパス2の間のフェーズエラー(Phase error)を生ずることがあり，これらのエラーはきわめて複雑でデバッグの難しいものであるから，十分に注意をする必要がある．

また，マクロ定義の中の入れ子になった条件付擬似命令から“EXITM”によってマクロ展開を中止した場合，エラーを生ずる場合があるので注意しなければならない．

これらの条件付擬似命令をまとめると，次のようになる．

IF1 (IF pass 1)

◇アセンブルのパス1ならば，以降のコードを生成する．

例：
```
IF1
    .PRINTX *Pass 1 Now !*
ENDIF
```

IF2 (IF pass 2)

◇アセンブルのパス2ならば，以降のコードを生成する．

例：
```
IF2
   .PRINTX  *Pass  2 Now !*
ENDIF
```

IF　　{式}
IFT　 {式}
COND {式}

◇式が0以外(真)ならば，以降のコードを生成する．

例：
```
X ASET  55
   IFT  X
       DB  'Yes'
   ENDIF
```
この場合，Xは0でないので DB 'Yes'のコードが生成される

IFF {式}
IFE {式}

◇式が0(偽)ならば，以降のコードを生成する．

例：

```
X  ASET  00
   IFF  X ………………   この場合，Xは0なので
      DB   'Yes'       DB 'Yes'のコードが生成される
   ENDIF ……………
```

IFDEF {シンボル}

◇シンボルが定義済か，または外部参照記号として定義されていれば，以降のコードを生成する．

例：

```
IFDEF MOJ1 ……………   MOJ1というモジュールが
       CALL MOJ1     定義されていれば，CALL MOJ1の
ENDIF ………………       コードを生成する
```

IFNDEF {シンボル}

◇シンボルが未定義で，また外部参照記号としても定義されていなければ，以降のコードを生成する．

例：

```
IFNDEF MOJ1 ……………   MOJ1という外部参照記号が
        EXTRN MOJ1    定義されていなければ，
ENDIF ………………        EXTRN MOJ1を定義する
```

IFB ＜変数＞

◇変数が空白ならば，以降のコードを生成する．

◇変数の両端の“＜　＞”は必ず必要である．

◇この条件付擬似命令は，マクロ定義のパラメータの有無のチェックに用いることが多い．

例：

```
VAL  MACRO      P1,P2,P3
     IFB   <P3> ………………   パラメータとしてP3がない場合，
         NOP                 NOPが生成される
     ENDIF ………………
       ∫
     ENDM
```

IFNB ＜変数＞

◇変数が空白でないならば，以降のコードを生成する．

◇変数の両端の“＜　＞”は必ず必要である．

◇この条件付擬似命令は，マクロ定義のパラメータの有無のチェックに用いることが多い．

例：

```
VAL  MACRO      P1, P2, P3
     IFNB  <P3>
           CALL  VAL1
     ENDIF
      ∫
     ENDM
```

パラメータとしてP3がある場合，CALL VAL1が生成される

IFIDN ＜式 1＞，＜式 2＞

◇式 1＝式 2 ならば，以降のコードを生成する．

◇変数の両端の“＜　＞”は必ず必要である．

IFDIF ＜式 1＞，＜式 2＞

◇式 1≠式 2 ならば，以降のコードを生成する．

◇変数の両端の“＜　＞”は必ず必要である．

ELSE

◇この条件付擬似命令は，IF＊＊のオプションで，IF＊＊の逆の条件が生じたときに，これ以降の ENDIF までのコードが生成される．

◇1 つの IF＊＊に対して，1 つの ELSE が許され，最も近い IF＊＊に対応する．

例：

```
IF1
     .PRINTX  *Pass 1*
ELSE
     .PRINTX  *Pass 2*
ENDIF
```

パス 1 で画面に
＊Pass 1＊と表示され，
パス 2 で画面に
＊Pass 2＊と表示される．

ENDIF/ENDC

◇条件付擬似命令の終了を定義する．

◇IF＊＊と ENDIF，COND と ENDC は，1 対 1 に対応していなければならない．

［補足］

MAC および RMAC(いずれもディジタル・リサーチ社のマクロアセンブラの名称)の場合には，これらの条件付擬似命令はもっと使用しやすくなっており，

```
IF　　条件
　　　　　　　[ 条件を満たすときの処理 ]
ENDIF
```

または，

```
IF　　条件
　　　　　　　THEN
　　　　　　　[ 条件を満たすときの処理 ]
　　　　　　　ELSE
　　　　　　　[ 条件に反するときの処理 ]
ENDIF
```

というような形になっている．条件として使用できるのは，

<	LS	
<=	LE	
=	EQ	（いずれを使用してもよい）
< >	NE	
>=	GE	
>	GR	

という，より一般的な条件判定文を書くことができ，MACRO−80よりも，使用しやすい形のものとなっている．

（E） マクロ関係擬似命令

この擬似命令には，ユーザが機能を定義するマクロ命令と，機能はすでに定義されていてパラメータだけを与えるマクロ命令の2種類のものがある．一般に，前者を“マクロ定義擬似命令”，後者を“ブロック擬似命令”と呼んでいる．ブロック擬似命令は単独で使用することもできるが，マクロ定義擬似命令の中で使用することもできる．このマクロ関係の擬似命令で注意しなければならないことは，この擬似命令を使用することによりソース・プログラムの大きさは，小さくすることができるが，この擬似命令がアセンブルされて，オブジェクト・コードに展開されると思わぬ大きさのオブジェクト・プログラムになってしまうので注意を要する．

これらのマクロ関係擬似命令は，すべて，次のような形式で使用される．

```
マクロ関係擬似命令(MACRO，REPなど)

　┌──────────────┐
　│ 展開される命令 │
　└──────────────┘

ENDM
```

すなわち，“ENDM(END of Macro)擬似命令”で終端された“マクロ関係擬似命令～ENDM”

というようなフォーマットになる．このENDM擬似命令は，マクロ関係擬似命令と必ず1対1の関係で対応がとれていなければならない．

また，このマクロ関係擬似命令には，ダミーパラメータをもつものが多いが，このダミーパラメータは，この擬似命令の中だけで使用される一時的なもので，記号(変数名，ラベル，予約語など)としては登録されない(このことにより，“できそうで，できないこと”ということがある．──→第2章参照)ので，AとかHLというようなレジスタ名と同じものでもよい．これらのマクロ関係擬似命令により，展開された部分(コード)は，リスト上で“+”記号が付けられる(アセンブル例参照のこと)．

また，このマクロ関係擬似命令には，その内部でのみ使用する特殊なマクロ演算子が用意されている．これらの演算子を使用することにより，これらのマクロ関係擬似命令の機能をさらに高めることができる．これらの演算子は他の部分では何の意味ももたない．

これらの擬似命令の使用法のテクニックに関しては，第2章で詳細に解説してあるので，そちらに譲ることにし，ここでは，その機能だけを次頁以下にまとめてみる．

◎ ブロック擬似命令に関するもの

REPT (REPeat)

{フォーマット} REPT {式}

◇REPT と ENDM の間の命令を式の回数だけ繰り返し生成する.

◇式の条件としては, 2バイトで表わされる範囲(0～FFFFH)でなければならない. また, 外部参照記号であったり, 未定義項目であったならばエラー(パス1とパス2のフェーズエラーに注意)となる.

例：

```
REPT    3
        NOP  →  NOP が3個生成される
ENDM
```

IRPC (Indefinite RePeat Character)

{フォーマット} IRPC {ダミー}, {文字列}

◇文字列から, 1文字(1バイト)ずつ取り出してダミーに代入する.

◇この命令は, 文字列の分解に使用できる.

例：

```
IRPC    REG, ABC              LD   A, 00
        LD     REG, 00   →    LD   B, 00   が生成される
ENDM                          LD   C, 00
```

IRP (Indefinite RePeat)

{フォーマット} IRP {ダミー}, {<パラメータ列>}

(パラメータ列を囲む“< >”は必ず必要)

◇パラメータ列から, 1パラメータずつ取り出してダミーに代入する.

◇IRPC 命令との相違は, ダミーに代入されるものが1文字単位ではなくて, 1パラメータ単位になるために複数の文字をダミーに代入できる.

例：

```
IRP     REG, <AF, BC, DE>      POP  AF
        POP    REG         →   POP  BC   が生成される
ENDM                           POP  DE
```

◎　マクロ定義擬似命令に関するもの

MACRO　(MACRO definition)

{フォーマット}　　　{マクロ名}　MACRO　[ダミーパラメータ]
　　　　　　　　　　(ダミーパラメータは，オプションである)

◇この擬似命令は，これまでのものと異なり，プログラムの中で一度だけ定義しておき，その後は，マクロ名だけを(パラメータのあるものは，パラメータを付ける)記述すれば，この命令によって定義されている部分のコードが展開される．

◇これまでの擬似命令は，それが現われたときにその機能を果たしたのに対して，この命令の場合には，あくまで“定義”であって，実際にプログラム中で使用されるのは，“マクロ名”の部分だけである．ゆえに，このマクロ名を使用する前にこの擬似命令によってマクロ名が定義されていなければならない．

◇このマクロ名を使用することを(MACRO 定義により定義されたコードを使用したいとき)“マクロ呼び出し(マクロ・コール)”という．

{マクロ呼び出し}

- マクロ呼び出しの形式は，次のとおりである．　{マクロ名}　[パラメータ]
- マクロ呼び出しのときに指定したパラメータが，そのマクロ名に対応するマクロ定義部のダミーパラメータに渡され，定義されたコード(命令)がその呼び出しを行なった部分に展開される(アセンブル例参照)．
- このマクロ呼び出しの部分も，また，マクロ定義の部分(マクロ名 MACRO．．の部分)も1行を越えてはならない．パラメータの数および長さはこの範囲で自由に設定してよい．
- マクロ定義部のダミーパラメータの数と，マクロ呼び出しのパラメータの数は異なってもよい．マクロ呼び出しのパラメータ数が，ダミーパラメータの数よりも少ないときには，ダミーパラメータには空(NULL)が入れられる．逆の場合には無視される．
- パラメータが複数ある場合には，それらは，カンマ，スペース，または，タブで区切られていなければならない．
- パラメータの一部として，カンマを含めたい場合には，山形のカッコで囲まなければならない．山形カッコの内部は，すべての項目を単一のパラメータとして扱われる．

　例：

　　マクロ名　A，B，C　←　3つのパラメータ“A”と“B”と“C”
　　マクロ名　<A，B，C>　←　1つのパラメータ“A，B，C”

- この擬似命令は入れ子にして使用できるが，1回の呼び出しで，1組の山形カッコがはずされるので多重呼び出しを行なう場合には注意をしなければならない．

{マクロ名}

- マクロ名の規則は，シンボルなどの名称とほぼ同じと考えればよいが，ソースプログラムが

アセンブルされるとき，パス1で，まずマクロ定義されているマクロ名をマクロテーブルに登録する．そして，それ以降は，名前に関するもの(ラベル，命令ニーモニック，擬似命令語など)をチェックする際に，まず，このマクロ・テーブルを調べる．すなわち，名前に関するものの中で，このマクロ名が最優先される．このことからマクロ名として命令ニーモニック(命令語)を名付けることもできる．

例：

```
PUSH    MACRO   REG
        POP     REG
        ENDM
```

というようなマクロ定義がしてあると，ソース・プログラム中の

```
        PUSH    レジスタ
```

は，すべて

```
        POP     レジスタ
```

に変わってしまう．

この性質を利用すれば，ザイログニーモニック(Z80)ともインテル・ニーモニック(i8080)とも異なる自分独自のニーモニックをつくることもできる．

LOCAL　(LOCAL label)

{フォーマット}　LOACL　{ラベル}，[，{ラベル}・・・]

◇この擬似命令は，MACRO定義の内部で使用するラベル(厳密にはすべてのシンボル)に対して，それがマクロ呼び出しにより展開されるたびに，異なるラベル名で展開することを定義するものである．

◇この擬似命令の使用により，マクロ展開によるラベルの二重定義をふせぐことができる．

◇このローカルなラベル名は，..0001から..FFFFの範囲でアセンブラが自動的に割り振ってくれる．ゆえに，それ以外で..nnnnという形のラベル名の使用は避けたほうがよい．

◇この擬似命令は，MACRO定義擬似命令の直後に使用する．

例：

```
TIMER   MACRO   NUM
        LOACL   LOOP   ← ローカルラベル定義
        LD      B, NUM
 LOOP:  NOP
 ↑      DJNZ    LOOP
 │      ENDM     ↑
 └───────────────┴──── マクロ呼び出しによって展開されるときには，
                       ..nnnnとなる．
```

◎ ブロック・マクロ定義に共通なもの

EXITM (EXIT Macro)

◇この擬似命令は，マクロ関係擬似命令(MACRO, REP など)から強制的に抜け出すもので，

GOTO ENDM

という機能と考えればよい.

◇マクロ関係擬似命令が入れ子になっている場合には，最も近い ENDM にとぶ.

例：

```
VAL     MACRO   STRING
        IFB     <STRING>
                PRINTX *ERROR*
                EXITM  ---------+
        ENDIF                   |  IFB が満たされると
        CALL    VALSUB          |  CALL VALSUB は展開されない
        ENDM    <---------------+
```

ENDM (END of Macro)

◇ブロックおよびマクロ定義擬似命令を終了する.

◎ マクロ関係擬似命令に使用するマクロ演算子

&

◇この演算子は，引用符で囲まれているダミーパラメータを展開する場合や，文字列とダミーパラメータを結合する場合に使用する．以下に例を示す．

<ダミーパラメータを引用符で囲む場合>

```
ERR     MACRO   X
        LD      A,'&X&'  ←X というダミーパラメータを
        JP      ERROR      引用符の中で使用している
        ENDM
```

これを，ERR A という形でマクロ呼び出しを行なうと，

```
        LD      A,'A'
        JP      ERROR
```

と展開される．& を付けずに，

```
        LD      A,'X'
```

という定義であると，同じ形でマクロ呼び出しを行なっても，

```
        LD      A,'X'
        JP      ERROR
```

となってしまい，パラメータの値が展開されない．

<文字列とダミーパラメータを結合する場合>

```
MSG     MACRO   X
        JP      MSG&X  ←文字列とダミーパラメータの結合
        ENDM
```

これを，MSG TEST という形でマクロ呼び出しを行なうと，

```
        JP      MSG&TEST
```

と展開される．この形の展開では，& が展開中に残っているがラベル・テーブル上ではMSGTEST と記録される．

；；

◇2つのセミコロンを付けたコメントは，マクロ呼び出しを行なっても展開されない．（リストファイルの容量を節約できる．）

！

◇この演算子は，それ以降に続く1文字を山形かっこで括ったのと同じ効果をもつ．

%

◇マクロのパラメータとして指定した式の値を返す．

これらの4つの演算子の中で，後者2つはあまり使用することはない．& 演算子の重要性は容易に理解できるであろう．ダブルセミコロンは，アセンブルの速度とリストファイルの大きさに関係してくるので，特別な場合以外は必ず使用したほうがよい．

（F）　その他の擬似命令

MACRO−80には，その他に以下のような擬似命令が用意されている．

INCLUDE　{ファイル名}
$INCLUDE　{ファイル名}
MACLIB　{ファイル名}

◇ファイル名で指定したソース・ファイルをこの命令の位置に挿入する．

◇ドライブ番号，ファイル拡張子を省略した場合，現在使用しているドライブから，「拡張子 .MAC」のファイルとみなす．

◇この擬似命令はマクロ定義の定義文の挿入としてプログラムの先頭に使用するとよい．

◇MS−DOS上のMASMとの互換性のためにはINCLUDEを使用するのがよいし，ディジタル・リサーチ社のMACおよびRMACとの互換性のためにはMACLIBを使用するのがよい．

例：　INCLUDE　MYLIB

ORG {式}

◇この擬似命令は，ロケーション・カウンタに値をセットする命令であるが，使用される場所によって機能が異なってくる．

◇ASEG擬似命令の後に使用された場合には，プログラムの開始アドレスを表わすことになる．

◇CSEGおよびDSEG擬似命令の後に使用された場合には，そのセグメントの開始点からの相対的な位置を表わすことになる．

◇また{式}の値は，パス1で既知でなければならない．

END　[{式}]

◇プログラムの終わりを指定する．

◇式を指定した場合には，リンクローダ（LINK−80）によって，100H番地からに“JMP　式”が自動的に挿入される．

.RADIX　{式}

◇{式}により基数を変更する．

◇基数としては2から16までの任意の基数を選択できる．初期値は10進数である．

例：

```
LD        A, 0FFH
.RADIX    16
LD        A, 0FF    ←  0FFHと同じ
```

.Z80

◇Z80のニーモニックを対象としてアセンブルする．この指定をした場合，ザイログ形式のZ80のニーモニックでプログラムを記述しなければならない(i8080の ニーモニックはエラーとなる)．

◇この指定は，MACRO−80の Z スイッチによっても行なうことができる．

.8080

◇i8080のニーモニックを対象としてアセンブルする．この指定をした場合，インテル形式のi8080のニーモニックでプログラムを記述する(Z80のニーモニックはエラーとなる)．

◇この指定は，MACRO−80の I スイッチによっても行なうことができる．

◇初期値は，このモードである．

3.2.4　アセンブルの方法

MACRO−80を使用してアセンブルを行なう場合，“MACRO−80の起動の方法”，“コマンドのフォーマット”，“コマンド・スイッチの使用法”の3つのことを理解していればよい．ここでは，これらのことについて3つに分けて解説する．

(A)　MACRO−80の起動の方法

MACRO−80を起動させるには2つの方法がある．その2つの方法の相違点は，MACRO−80の起動コマンドである“M80”につづいて作業内容を表わすコマンドを入力する方法と，MACRO−80を起動させた後に，MACRO−80が返してくるプロンプトに対してコマンドを入力する方法である．これは，CP/Mの標準コマンドであるPIPの場合と同じようなものと考えればよい．

```
方法1　　A>M80　コマンド　（改行）
方法2　　A>M80　（改行）
　　　　 ＊コマンド（改行）
　　　　 ＊^C
```

方法1の場合のメリットは，MACRO−80のコマンドラインに作業内容を表わすコマンドを記述できるので，SUBMITコマンドを使用したバッチ処理が可能となる(XSUBコマンドを使用すれば，方法2を使用してもバッチ処理を行なうことができる)．

方法2の場合のメリットは，一度に複数のソース・プログラムのアセンブルを行なうことができるということにある．すなわち，PIPコマンドの場合と同じように，この方法でMACRO−80を起動させた場合には，コントロールCをキーインするまでMACRO−80のコマンドモードから脱出しない．それゆえ，次々にコマンドを与えることにより複数のソース・プログラムをアセンブルすることが可能になるわけである．

(B) コマンドのフォーマット

MACRO−80のコマンドは，“=”記号の左右でソースとオブジェクトを分離している．すなわち，PIPコマンドと同じように“=”記号の左側が新しく生成するファイルの仕様を表わし，右側がソース・ファイルの仕様を表わす．すなわち，

新しくつくり出されるファイルの仕様＝ソース・ファイルの仕様．

という形になる．ここでソース・ファイルの仕様とは，そのソース・ファイルが記録されているドライブ番号とファイル名を意味する．これはCP/M上のファイルを表現するときのルールとなんら変わるところはない．ただし，拡張子に関しては，省略した場合には，MACRO−80のソース・ファイルのデフォルト値である“.MAC”が採用される．

例：	ドライブBにあるTEST.MACをアセンブルする場合の ソース・ファイルの仕様の表現法 =B: TEST　　または　=B：TEST.MAC

“=”記号の左側の“新しくつくり出されるファイルの仕様”とは，カンマ“,”を区切りとして，

オブジェクト・ファイル仕様，リスト・ファイル仕様．

という形になる．この場合のファイル仕様というのも“ドライブ番号：ファイル名”というCP/M上のファイル表現の標準型となっている．ただし，拡張子を省略した場合には，オブジェクト・ファイルは“.REL”，リスト・ファイルは“.PRN”が採用される．

例：	ドライブBにTTOBJ.RELというオブジェクト・ファイルと， TTLST.PRNというリスト・ファイルをつくる場合 B: TTOBJ, B:TTLST=ソース・ファイル仕様

ソース・ファイルとドライブ番号およびファイル名が同じでよいならば，この“=”記号の左側は完全に省略できる．

例：	>M80　=B:TEST ドライブBにTEST.RELが作成される． （リスト・ファイルも必要なときは，スイッチ“/L”を付加する）

エラー・チェックなどのために，オブジェクト・ファイル，リスト・ファイルともに不用な場合には“=”記号の前にコンマを置けばよい．

> 例： ＞M80 ，=B：TEST
>
> アセンブルを行ない，エラー・チェックは行なうが，オブジェクト・ファイルおよびリスト・ファイルは作成されない．

(C) コマンド・スイッチの使用法

コマンドにこのスイッチを付加することによって出力ファイルの制御を行なったり，ソース・ファイルのニーモニックがZ80のものか，i8080のものかを指定することができる．これらのスイッチは，コマンドの“=”記号の左側に付けても，右側に付けてもよい．また，各スイッチの前には“/”を付けなければならない．

スイッチ	機　　　能
Z	ソース・ファイルが，Z80ニーモニックで記述されていることをアセンブラに指示する．ソース・ファイル内で“.Z80”が使用されている場合には，必要ない．
I	ソース・ファイルが，i8080のニーモニックで記述されていることをアセンブラに指示する．デフォルト値はこのモードである．
L	コマンドに関係なく，リスト・ファイルを出力する．
C	コマンドに関係なく，リスト・ファイルに換えてクロスリファレンス・ファイルを出力する．注）LとCは両方を選択することはできない．
R	コマンドに関係なく，オブジェクト・ファイルを出力する．
O	リスト内のすべての数値を8進数で表示する．
H	リスト内のすべての数値を16進数で表示する．デフォルト値はこのモードである．
P	アセンブル時に使用するスタック領域を225バイト追加する．アセンブル時にスタック・オーバフローが起こったときにのみ使用する．
M	DS擬似命令によって定義されたエリアをゼロクリアする．これを行なわない場合，このデータ・エリアの内容は保証されない．
X	条件付擬似命令の偽条件のリスト出力を停止する．TFCOND擬似命令との関係に注意しなければならない．

3.2.5　アセンブルの実例とエラーメッセージ

(A)　アセンブルの実例

16 ビットの乗算と除算を行なう外部モジュールをマクロ定義したメインプログラムでテストするプログラムのアセンブルを行なう．マクロ定義部分は外部にマクロ定義ライブラリとして用意し，メインプログラムからは，INCLUDE 擬似命令によって呼び出している．

◇メインプログラム部分

```
        TITLE       Main
        INCLUDE     MACLIB.LIB    ← 外部マクロ定義部の読み込み
        .REQUEST    SUB           ← 外部モジュールとのリンク指定
        .Z80    ← Z80ニーモニック使用宣言

  START::  <---------------------------------- 共通参照記号
        LD          HL,10                   |
        MUL16       HL,2        ← MUL16マクロ呼び出し
        DIV16       HL,4        ← DIV16マクロ呼び出し
 EXIT::   <----------------------------------
        JP          0
        END         START       ←リンクローダへの開始アドレスの指示
```

注意）外部モジュールは，ライブラリマネジャを使用してSUBというライブラリにしてある．

◇アセンブルは次のように 行なう

```
A>M80  =MAIN/L       ← MAIN.MACをアセンブルして，MAIN.RELとMAIN.PRNを得る
No Fatal error(s) ← アセンブルのエラーはない
A>M80  =M16/L        ← M16.MACをアセンブルして，M16.RELとM16.PRNを得る
No Fatal error(s) ← アセンブルのエラーはない
A>M80  =D16/L        ← D16.MACをアセンブルして，D16.RELとD16.PRNを得る
No Fatal error(s) ← アセンブルのエラーはない
A>
```

◇マクロ定義ライブラリ部分

```
;     Macro Lib  No.1

MUL16        MACRO      P1,P2
             EXTRN      M16
             ifdif      <P2>,<HL>
             ifdif      <P1>,<HL>
             LD         HL,P1
             endif
             endif
             ifdif      <P2>,<DE>
             LD         DE,P2
             endif
             CALL       M16
             ifdif      <P1>,<HL>
             LD         P1,HL
             endif
             ENDM

DIV16        MACRO      P1,P2
             EXTRN      D16
             ifdif      <P2>,<HL>
             ifdif      <P1>,<HL>
             LD         HL,P1
             endif
             endif
             ifdif      <P2>,<DE>
             LD         DE,P2
             endif
             CALL       D16
             ifdif      <P1>,<HL>
             LD         P1,HL
             endif
             ENDM
```

MUL16のマクロ定義部（EXTRN M16 ～ endif）

DIV16のマクロ定義部（EXTRN D16 ～ endif）

注意） CALL文で使用しているモジュール名をEXTRN宣言していることに注意．

◇乗算モジュール　　<u>HL←HL×DE</u>

```
        TITLE   Modul1
        NAME    ('M16')    ←ライブラリマネジャへのモジュール名
        PUBLIC  M16        ←共通参照記号宣言
        .Z80                モジュール名(サブルーチン)
M16:
        LD      C,L        ┐
        LD      B,H        │
        LD      HL,0       │ 処理部分
        LD      A,15       │
J1:                        │
        SLA     E          │
        RL      D          │
        JR      NC,J2      │
        ADD     HL,BC      │
J2:                        │
        ADD     HL,HL      │
        DEC     A          │
        JR      NZ,J1      │
        OR      D          │
        RET     P          │
        ADD     HL,BC      │
        RET                ┘
        END
```

◇除算モジュール　　HL←HL÷DE

```
        TITLE   Modul2
        NAME    ('D16')     ←ライブラリマネジャへのモジュール名
        PUBLIC  D16         ←共通参照記号宣言
        .Z80                 モジュール名(サブルーチン)
D16:
        LD      A,E
        OR      D
        JR      NZ,J1       処理部分
        LD      HL,0
        LD      D,H
        LD      E,L
        SCF
        RET
J1:
        LD      C,L
        LD      A,H
        LD      HL,0
        LD      B,16
        OR      A
L1:
        RL      C
        RLA
        RL      L
        RL      H
        PUSH    HL
        SBC     HL,DE
        CCF
        JR      C,L2
        EX      (SP),HL
L2:
        INC     SP
        INC     SP
        DJNZ    L1
        EX      DE,HL
        RL      C
        LD      L,C
        RLA
        LD      H,A
        OR      A
        RET
        END
```

◇メインプログラムのアセンブルリスト

```
Main  MACRO-80 3.44      09-Dec-81          PAGE      1
↑              ↑
└タイトル       └ バージョン番号              TITLE     Main

INCLUDE擬似命令に   C                       INCLUDE   MACLIB.LIB
より読み込まれた    C     ;                 Macro Lib  No.1
マクロ定義部分 →    C      MUL16            MACRO     P1,P2
                    C                       EXTRN     M16
                    C                       ifdif     <P2>,<HL>
                    C                       ifdif     <P1>,<HL>
                    C                       LD        HL,P1
                    C                       endif
                    C                       endif
                    C                       ifdif     <P2>,<DE>
                    C                       LD        DE,P2
                    C                       endif
                    C                       CALL      M16
                    C                       ifdif     <P1>,<HL>
                    C                       LD        P1,HL
                    C                       endif
                    C                       ENDM
                    C
                    C      DIV16            MACRO     P1,P2
                    C                       EXTRN     D16
                    C                       ifdif     <P2>,<HL>
                    C                       ifdif     <P1>,<HL>
                    C                       LD        HL,P1
                    C                       endif
                    C                       endif
                    C                       ifdif     <P2>,<DE>
                    C                       LD        DE,P2
                    C                       endif
                    C                       CALL      D16
                    C                       ifdif     <P1>,<HL>
                    C                       LD        P1,HL
                    C                       endif
                    C                       ENDM

                    .REQUEST  SUB ← リンクローダへの指示
                    .Z80            アセンブルの時点では
                                    オブジェクトには反映
                                    されない
```

```
0000'                         START::
0000'   21 000A                   LD        HL,10
                                  MUL16     HL,2
0003'   11 0002          +        LD        DE,2     ─┐ MUL16が
0006'   CD 0000*         +        CALL      M16    ───┘ マクロ展開された部分
                                  DIV16     HL,4
0009'   11 0004          +        LD        DE,4     ─┐ DIV16が
000C'   CD 0000*         +        CALL      D16    ───┘ マクロ展開された部分
000F'                         EXIT::
000F'   C3 0000                   JP        0

                                  END       START ← アセンブル時には
                                                      反映しない

Main           MACRO-80 3.44       09-Dec-81          PAGE      S

Macros:
DIV16          MUL16          ← 使用されているマクロ名

Symbols:
000D*          D16       000FI'     EXIT        0007*     M16
0000I'         START

     ↑                       ┌─ * マークは外部参照記号
     └──── 使用されたシンボル ├─ I マークは共通参照記号
                             └─ ' マークは相対番地をあらわす

No Fatal error(s)
```

◇乗算モジュールのアセンブルリスト

```
Modul1          MACRO-80 3.44       09-Dec-81           PAGE      1

                                        TITLE     Modul1
                                        NAME      ('M16')
                                        PUBLIC    M16
                                        .Z80
0000'                               M16:
0000'     4D                            LD        C,L
0001'     44                            LD        B,H
0002'     21 0000                       LD        HL,0
0005'     3E 0F                         LD        A,15
0007'                               J1:
0007'     CB 23                         SLA       E
0009'     CB 12                         RL        D
000B'     30 01                         JR        NC,J2
000D'     09                            ADD       HL,BC
000E'                               J2:
000E'     29                            ADD       HL,HL
000F'     3D                            DEC       A
0010'     20 F5                         JR        NZ,J1
0012'     B2                            OR        D
0013'     F0                            RET       P
0014'     09                            ADD       HL,BC
0015'     C9                            RET
                                        END

Modul1          MACRO-80 3.44       09-Dec-81           PAGE      S
Macros:
Symbols:
0007'           J1          000E'   J2          0000I'      M16
                                                     ↑
No Fatal error(s)                                共通参照記号マーク
```

◇除算モジュールのアセンブルリスト

```
Modul2          MACRO-80 3.44        09-Dec-81            PAGE       1

                                     TITLE     Modul2
                                     NAME      ('D16')
                                     PUBLIC    D16
                                     .Z80
0000'                            D16:
0000'     7B                         LD        A,E
0001'     B2                         OR        D
0002'     20 07                      JR        NZ,J1
0004'     21 0000                    LD        HL,0
0007'     54                         LD        D,H
0008'     5D                         LD        E,L
0009'     37                         SCF
000A'     C9                         RET
000B'                            J1:
000B'     4D                         LD        C,L
000C'     7C                         LD        A,H
000D'     21 0000                    LD        HL,0
0010'     06 10                      LD        B,16
0012'     B7                         OR        A
0013'                            L1:
0013'     CB 11                      RL        C
0015'     17                         RLA
0016'     CB 15                      RL        L
0018'     CB 14                      RL        H
001A'     E5                         PUSH      HL
001B'     ED 52                      SBC       HL,DE
001D'     3F                         CCF
001E'     38 01                      JR        C,L2
0020'     E3                         EX        (SP),HL
0021'                            L2:
0021'     33                         INC       SP
0022'     33                         INC       SP
0023'     10 EE                      DJNZ      L1
0025'     EB                         EX        DE,HL
0026'     CB 11                      RL        C
0028'     69                         LD        L,C
0029'     17                         RLA
002A'     67                         LD        H,A
002B'     B7                         OR        A
002C'     C9                         RET

                                     END

Modul2          MACRO-80 3.44        09-Dec-81            PAGE       S

Macros:
Symbols:
0000'           D16        000B'     J1           0013'     L1
0021'           L2
          共通参照記号マーク
No Fatal error(s)
```

(B) MACRO−80 のエラーメッセージ

≪アセンブル・リスト中に示されるエラーコード≫

A	(Argument error) 擬似命令に対する式フィールドの形式が正しくない.
C	(Conditional nesting error) ENDIF のない条件付擬似命令や，ELSE の使用法に誤りがある.
D	(Double Defind symbol) 参照したシンボルが二重定義になっている.
E	(External error) 外部参照記号が誤って使用されている.
M	(Multiply Defined symbol) 定義しようとしたシンボルはすでに定義済である.
N	(Number error) 数値の表現に誤りがある.
O	(Bad opecode or objectionable syntax) ・オペコードがまちがっているか，式フィールドのフォーマットに誤りがある. ・ENDM や LOCAL がマクロ定義の外にある. ・SET，EQU，MACRO にシンボル名やマクロ名が指定されてない.
P	(Phase error) ラベル，または EQU のシンボル名の値がパス 1 とパス 2 とで異なる.
Q	(Questionable) 行が正しく終了していない場合などで，警告エラーである.
R	(Relocation) 式フィールドのリロケーションをまちがって使用した場合など.

U	(Undefined symbol) シンボルの未定義.
V	(Value error) パス1で決まる値をもたなければならない擬似命令(.PAGE, DS, IF, IFE)が定義されていない値をもっている.

≪アセンブル・リストの終わりに表示されるエラーメッセージ≫

%No END statement	END文がない. またはEND文が偽条件ブロックやマクロ定義内で定義されている.
Unterminated conditional	条件ブロックが終了してない.
Unterminated REPT/IRP/IRPC/ MACRO	ブロック擬似命令のブロックが終了していない.
Symbol table full	シンボルテーブルの作成中にメモリが不足した.

≪エラーメッセージに対する注意≫

◇Rエラーコードが発生した場合 これは, 式内で, アブソリュート(絶対)値からリラティブ(相対)値を引いた場合などに生ずる.
◇UエラーとVエラーの関係 パス1でVエラーが表示された場合には, パス2でUエラーが表示される.
◇Symbol table full エラーが発生した場合 マクロ定義が多い場合に発生することが多い. これは, マクロ定義テーブルも, このシンボルテーブル内に取られるために生ずる. マクロ定義の数を少なくするか, マクロ定義内のコメントの前のセミコロン(;)をダブルセミコロン(;;)に変えてみる.

3.3　クロスリファレンス・ユーティリティ(CREF−80)の使用法

MACRO−80の"/C"スイッチにより出力されたクロスリファレンス・ファイルからクロスリファレンス・リスト(前後参照リスト)を作成するのがこの"クロスリファレンス・ユーティリティ(CREF−80)"である．簡単なプログラムの場合にはあまり必要ないが，変数やラベルを数多く使用しているようなプログラムの場合には，それらの使用状況を把握するのにきわめて便利な機能である．

このユーティリティの使用方法はきわめて簡単で，次のようにすれば MACRO−80 から出力されたクロスリファレンス・ファイルから，クロスリファレンス・リスト・ファイルを得ることができる．

例：

◎　ソース・ファイル(クロスリファレンス・ファイル)と同じ名称のクロスリファレンス・リスト・ファイルをつくる場合

A>CREF80　=ソース・ファイル名

◎　異なるファイル名のクロスリファレンス・リスト・ファイルをつくる場合

A>CREF80　ファイル名 = ソース・ファイル名

これらのファイルの拡張子を省略した場合，ソース・ファイルは".CRF"，クロスリファレンス・リスト・ファイルは".PRN"となる．この操作によって作成されるクロスリファレンス・リストと一般のリスト・ファイルとの相違点は，次のとおりである．

- 各ステートメント(行)には行番号が付けられる．
- リストの最後に変数名が，定義されている行番号と，その変数が使用されている行番号を伴ってアルファベット順に表示される．

このクロスリファレンス・リストの例を，次に示す．

◇クロスリファレンス・リストの例

アセンブルの実例で使用したMAINプログラムを，クロスリファレンスがとれるように再アセンブルし，クロスリファレンス・ユーティリティCREF-80によってクロスリファレンス・リストを作成してみる．

《操　作》

```
A>M80  ,=MAIN/C            ← MAIN.MACをアセンブルして，MAIN.CRFを得る
No Fatal error(s)          ← アセンブルのエラーはない
A>CREF80  =MAIN            ← CREF80を使用して，MAIN.CRFからクロスリファレンス・
                              リストMAIN.PRNを得る
A>TYPE  MAIN.PRN           ← クロスリファレンス・リストの表示
```

```
Main  MACRO-80 3.44         09-Dec-81             PAGE      1

  1                                                       TITLE     Main
  2   行番号
  3                                   C                   INCLUDE   MACLIB.LIB
  4                                   C        ;          Macro Lib  No.1
  5                                   C
  6                                   C        MUL16      MACRO     P1,P2
  7                                   C                   EXTRN     M16
  8                                   C                   ifdif     <P2>,<HL>
  9                                   C                   ifdif     <P1>,<HL>
 10                                   C                     LD      HL,P1
 11                                   C                   endif
 12                                   C                   endif
 13                                   C                   ifdif     <P2>,<DE>
 14                                   C                     LD      DE,P2
 15                                   C                   endif
 16                                   C                   CALL      M16
 17                                   C                   ifdif     <P1>,<HL>
 18                                   C                     LD      P1,HL
 19                                   C                   endif
 20                                   C                   ENDM
 21                                   C
 22                                   C        DIV16      MACRO     P1,P2
 23                                   C                   EXTRN     D16
 24                                   C                   ifdif     <P2>,<HL>
 25                                   C                   ifdif     <P1>,<HL>
 26                                   C                     LD      HL,P1
 27                                   C                   endif
 28                                   C                   endif
 29                                   C                   ifdif     <P2>,<DE>
```

```
30                              C               LD      DE,P2
31                              C             endif
32                              C             CALL      D16
33                              C             ifdif     <P1>,<HL>
34                              C               LD      P1,HL
35                              C             endif
36                              C             ENDM
37                              C
38                                            .REQUEST  SUB
39                                            .Z80
40        0000'                        START::
41        0000'    21 000A                    LD        HL,10
42                                            MUL16     HL,2
43        0003'    11 0002       +              LD      DE,2
44        0006'    CD 0000*      +            CALL      M16
45                                            DIV16     HL,4
46        0009'    11 0004       +              LD      DE,4
47        000C'    CD 0000*      +            CALL      D16
48        000F'                        EXIT::
49        000F'    C3 0000                    JP        0
50
51                                            END       START

Main  MACRO-80 3.44      09-Dec-81        PAGE      S

Macros:
DIV16           MUL16

Symbols:
000D*           D16             000FI'        EXIT          0007*
M16
0000I'          START

No Fatal error(s)
```

```
D16              46#      47
DIV16            22#      45
EXIT             48#
M16              43#      44
MUL16             6#      42
START            40#      51
```

《前後参照リスト》
#マークは，その行番号で定義されていることを示している．無印の番号は，そのシンボルを使用している行番号を示す．

3.4 リンクローダ(LINK−80)の効果的な使用法

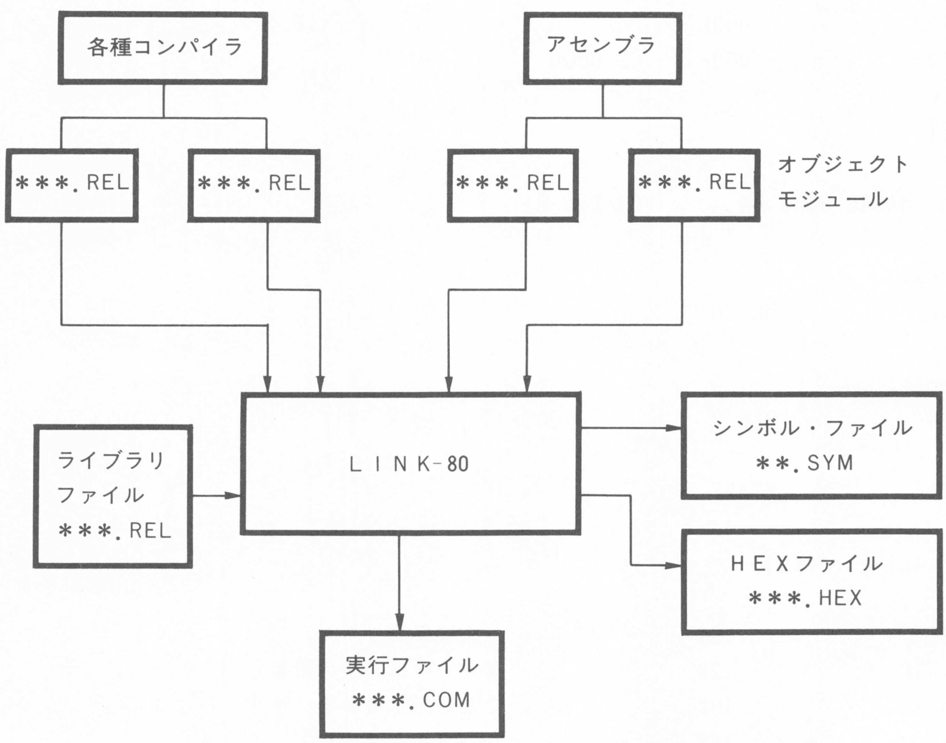
この節では，リンクローダとして，マイクロ・ソフト社の LINK−80 の使用法を，次の手順で解説する．

3.4.1 リンクローダの機能

3.4.2 リンクの方法

3.4.3 リンクの実例とエラーメッセージ

◎ LINK-80の機能



3.4.1　リンクローダの機能

リンクローダ(LINK−80)には，次の4つの大きな機能がある．

- オブジェクト•モジュール(***REL ファイル)を実行可能なロード•モジュール(実行ファイル ***.COM)に変換する．
- 個々に開発したオブジェクト•モジュールを1つにまとめる．
- ライブラリ•ファイルから必要なファイルを探してリンクする．
- プログラム•エリアとコード•エリアのロード•アドレスを決定する．

これらの機能は，リンクローダのコマンドとスイッチの組み合わせにより選択することができる．このリンクローダの機能というのは，その名前が示すとおり，

> リンク(結合)して，ロードするもの．

と考えればよい．CP/M の標準アセンブラである ASM の場合の“LOAD コマンド”と同様な機能をもっているわけである．しかし，このリンクローダが LOAD コマンドと異なることは，複数のオブジェクト•プログラムを1つのプログラムにまとめる機能があるという点である．この複数のオブジェクト•プログラムというのは，リロケータブル•オブジェクト•モジュール(一般に“***.REL”のファイル)の形をしたものであればなんでもかまわない．このリロケータブル•オブジェクト•モジュールを出力するものとしては，マイクロ•ソフト系のコンパイラ言語をはじめとして，CP/M 上のほとんどのコンパイラ言語をあげることができる．このことから，このコンパイラからのオブジェクト•プログラムと，MACRO−80により作成したアセンブラからのオブジェクト•プログラムをリンク(結合)することもできるわけである．また，ライブラリ•ファイルの中からメインプログラムで必要とするモジュールのみを選択して結合する機能もある．

この LINK−80 での“リンク”という作業はすべてメモリ上で行なわれる．それゆえ，リンクできるプログラムのサイズというのは，その CP/M システムが有しているメモリのサイズにより決定される．また，オーバーレイなどの機能は有してない．すなわち，この LINK−80 というリンクローダは“リンク＆ロード”ということに対する最も基本的な機能だけをもったものということができる(より高度なリンクローダとしては，PLINK−II などがある)．

この LINK−80 でのリンクのされ方は，“ASEG 擬似命令”を使用したプログラムと，“CSEG•DSEG 擬似命令”を使用したプログラムとでは，かなり相違がある．“ASEG 擬似命令”を使用したプログラムの場合には，LOAD コマンドとほぼ同じと考えてもよい．しかし，“CSEG•DSEG 擬似命令”を使用したプログラムの場合には，LOAD コマンドとは全く異なり，103 H 番地から，まず DSEG で指定したデータがロードされる．

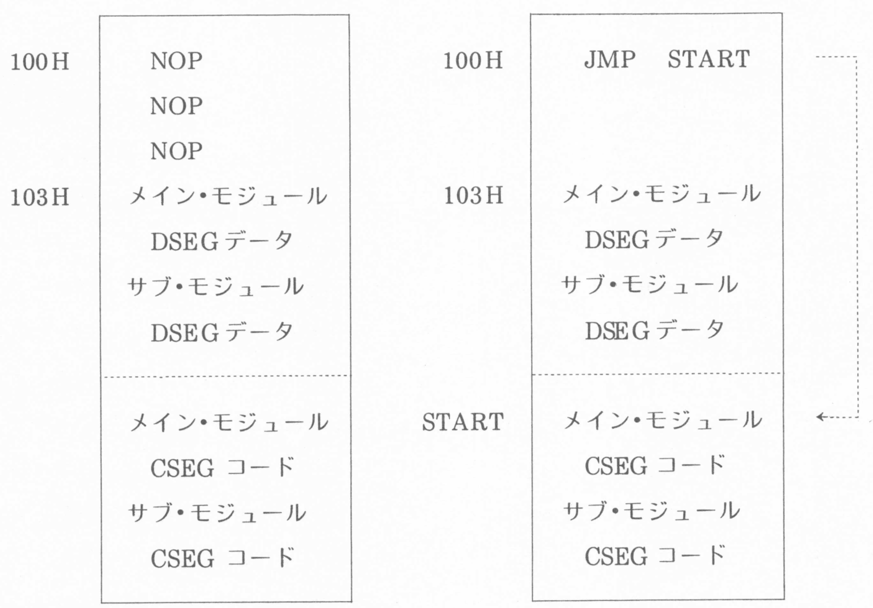
次に，その続きに CSEG で指定したコードがロードされる．では，100 H〜102H はどうなる

かというと，“END 擬似命令”で“式”が指定されてない場合には，3つのNOP命令が入る．であるから，このままの形で実行ファイルを作成して実行すると，当然，暴走してしまう(データ・エリアからプログラムを実行することになる)．

しかし，“END 擬似命令”で“式”が指定してあると，3つのNOP命令の代わりに“JMP 式”というコードを自動的に生成してくれる．それゆえ，この“CSEG・DSEG 擬似命令”を使用したリロケータブルなプログラムをリンクする場合には，“END 擬似命令”がきわめて重要な働きをするわけである．であるから，このようなプログラムの場合，そのメイン・モジュールの“END 擬似命令”には，必ず“式”を指定するようにしなければならない．これは，アセンブルの際も，また，リンクの際も気づかないことが多いので，十分に気をつけなければならない．

◎ CSEG・DSEG を使用した場合のリンク時のメモリ・マップ

<ENDだけの場合>　　<END 式の場合>　式=STARTと仮定



3.4.2 リンクの方法

LINK-80を使用してオブジェクト・プログラムのリンクを行なう場合，“LINK-80の起動の方法”，“LINK-80のコマンドのフォーマット”，“コマンド・スイッチの使用法”の3つのことを理解していればよい．ここでは，これらのことについて解説する．

(A) LINK-80の起動の方法

LINK-80を起動させるには2つの方法がある．その2つの方法の相違点は，LINK-80の起動コマンドである“L80”につづいて作業内容を表わすコマンドを入力する方法と，LINK-80を起

動させた後に，LINK−80が返してくるプロンプトに対してコマンドを入力する方法である．これは，CP/M の標準コマンドである PIP の場合と同じようなものと考えればよい．

```
方法 1    A>L80       コマンド   (改行)
方法 2    A>L80       (改行)
          *コマンド   (改行)
             ∫
          */E         (改行)      ←  LINK−80からの脱出
```

方法1の場合のメリットは，LINK−80のコマンドラインに作業内容を表わすコマンドを記述できるので，SUBMIT コマンドを使用したバッチ処理が可能となる(XSUB コマンドを使用すれば，方法2を使用してもバッチ処理を行なうことができる)．

方法2の場合のメリットは，使用しているラベルの状態や，メモリの使用状況などを確認しながらリンク作業を行なうことができるという点にある．また，ソース•プログラム中の“.REQUEST 擬似命令”によりリンクされたモジュールもこの方法2のやりかたでなければ確認することができない(リンクの実例を参照のこと)．これらのことより，簡単なリンクとロード(CP/M の LOAD コマンドに対応する処理)だけならば，方法1のコマンド•ライン上へのコマンドの記述による方法でよいが，一般的には，方法2を使用したほうがよい．

(B) コマンドのフォーマット

LINK−80のコマンドは，“ファイル仕様[/スイッチ]”が基本になっている．

すなわち，リンクという作業においては，いくつかのオブジェクト•ファイルをメモリ上に展開しなければならない．このオブジェクト•ファイルを指定する作業が一番目の作業となる．すなわち，リンクローダからのプロンプトに対して

*ファイル名，ファイル名，•••••

と答えることにより，指定したオブジェクト•ファイルが，その順番でメモリ上に展開される．これらは，入力ファイルと考えてもよい．リンクローダは，特別なスイッチ(/N:Name)を付けない限り入力ファイルとみなす．メモリ上に展開されたプログラムを，実行ファイルとして出力するときは，Name スイッチ(/N)をファイル名に付加する．これにより，リンクローダは，そのファイルが出力ファイルであることを知り，それに対応する処理を行なう．また，このファイル名の拡張子を省略した場合，入力ファイルは，“.REL”というリロケータブル•オブジェクト•ファイルを選択する．また，出力ファイルの場合には特別な指定(/X による HEX ファイル指定)をしない限り，“.COM”という実行ファイルになる．リンクローダのコマンドは，単にファイル名を指定するだけの機能であり，次のスイッチと組み合わせて意味をもつものである．

(C)　コマンド・スイッチの使用法

コマンドにこのスイッチを付加することによって出力ファイルの制御を行なったり，入力ファイルの中から必要なモジュールのみを選択してリンクするというようなことができる．これらのスイッチの前には必ず/(スラッシュ)を付加しなければならない．

このスイッチの使用法に関しては，その機能だけをまとめておくので，詳細は「3.3.3 リンクの実例とエラーメッセージ」を参照してほしい．

LINK-80のコマンド・スイッチ

スイッチ	機　　能
N (NAME)	出力ファイル名を指定する． ファイル名/Nにより，/Eが実行された時点，または/Gが完了した時点で現在メモリ中にあるプログラムを{ファイル名.COM}という実行形式のファイルとしてディスクにセーブする． 例：　B：TEST/N
E (END)	LINK-80から脱出してCCPの管理下にもどる． .REQUEST擬似命令などが指定されていれば，この時点でそれらのファイルから未定義記号を調べる．Nameスイッチが指定してあれば，メモリ上のプログラムをディスクにセーブする．
E:名前	これは，Endスイッチのオプションで，100H番地に“JMP 名前”というコードをいれるものである．この場合の名前は，PUBLIC宣言がしてあるものでなければならない．END擬似命令で式が指定してある場合にはこのオプションは必要ない． 例：　/E：START
G (GO)	これはEスイッチとその機能はほとんど同じであるが，CCPにもどるまえにプログラムを実行する．
G:名前	実行の開始番地を指定したGスイッチ．
PおよびD	入力ファイルのロード開始番地を指定する．
(PROGRAM) (DATA)	/Pは，プログラム領域の開始番地を指定し，/Dは，データ領域の開始番地を指定する．主にROM化するプログラムの場合に使用する． このスイッチを使用する場合には，END擬似命令で式を指定してはならない．このスイッチは，ASEG擬似命令を使用したプログラムには無意味な指定となる． 例：プログラムを200H番地，データを300H番地からに設定する場合 　　*/P:200，/D:300
M (Mark)	プログラム中で使用されている外部参照記号の一覧表を表示する．未定義の記号には，記号の前に“-”を表示する(v3.44)．
U (UFO)	プログラム中の未定義の記号を表示する．
S (Search)	指定されたファイルから，未定義の外部参照記号をサーチする． ファイル名/Sで，そのファイル(一般にライブラリ・ファイル)から，その時点で未定義となっているモジュールを探してリンクする(リンク実例参照)．

R (Reset)	リセット．それまでのリンク作業をすべて初期状態にもどす． 誤ってまちがったファイルをリンクしてしまったときに使用する．
Y	ZSID 用のシンボル・ファイル(***．SYM)を作成する．
X	実行ファイル(***．COM)の代わりに HEX ファイル(***．HEX)を作成する．

3.4.3　リンクの実例とエラーメッセージ

(A)　リンクの実例

“4.2.5 アセンブルの実例” で示した MAIN プログラムを，リンクローダを使用して 2 通りの方法で実行ファイルを作成してみる．ただし，乗算モジュールと除算モジュールはライブラリマネジャを使用して “SUB.REL”というライブラリにしてある(これに関しては，“4.5.3 ライブラリ化の実例” を参照のこと)．

≪操　作≫

1.　コマンドラインに操作を記述する方法

A>L80　MAIN，MAIN/N/Y/E　(改行)←コンソールからのキー入力

- E：終了
- Y：SYM ファイル作成
- N：COM ファイル作成(ネーム)
- MAIN (2番目)：COM ファイルの名前
- MAIN (1番目)：リンクするファイル名

システム応答：

Link-80　3.44　09-Dec-81　Copyright(c) 1981　Microsoft

Data　　0103　0158　　< 85 >

- 0103：開始アドレス
- 0158：終了アドレス
- < 85 >：使用メモリ数

40900 Bytes Free　← 残り使用可能メモリ数

[0103　　0158　　　1]　← COM ファイルの仕様

- 0103：開始アドレス
- 0158：終了アドレス
- 1：使用ページ数

A>　ここまでで “MAIN.COM” と “MAIN.SYM” という 2 つのファイルが作成される．

2. LINK−80 を起動させプロンプトに応える方法

A＞L 80 （改行）←コンソールからのキー入力

Link−80 3.44 09−Dec−81 Copyright (c) 1981 Microsoft

*MAIN （改行） リンクするファイル名

Data 0103 0115 < 18 > ←MAIN. REL のみ

使用メモリ数（18）

最終アドレス（0115）

開始アドレス（0103）

SUB REQUEST ←MAIN. RELの中で SUB. REL のリンクを要求している.

−D16 0110 −M16 010A

M16が使用されているアドレス（010A）

D16 が使用されているアドレス（0110）

2 Underfined Global (s) ←この時点では上記 2 つのシンボルが未定義となる.

（この時点では SUB. REL はロードされていない）

40967 Bytes Free ←この時点での残り使用可能メモリ数

*MAIN/N/Y/E （改行） COM ファイルの名前をコンソールからキー入力

[ここで SUB. REL がロードされる]

Data 0103 0158 < 85 >

MAIN. COM の使用メモリ数（85）

MAIN. COM の終了アドレス（0158）

MAIN. COM の開始アドレス（0103）

40900 Bytes Free ←残り使用可能メモリ数

[0103 0158 1] ←COM ファイルの仕様

使用ページ数（1）

終了アドレス（0158）

開始アドレス（0103）

A＞ ここまでで“MAIN. COM”と“MAIN. SYM”という 2 つのファイルが作成される.

注意） 2番目の方法でリンクした場合には，実行ファイルの指定をした時点で REQUEST に対するリンクが行なわれる.

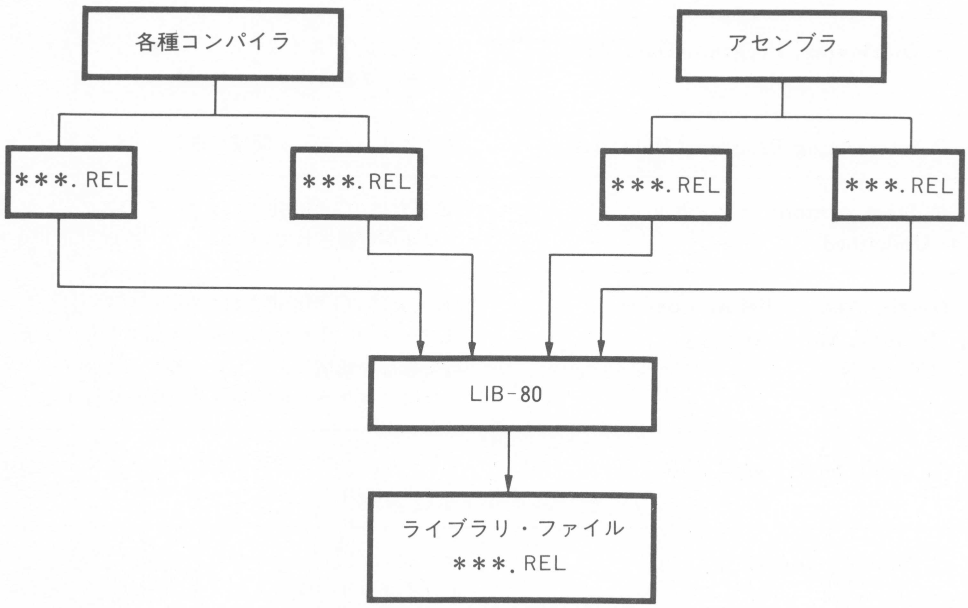
(B)　LINK-80 のエラーメッセージ

? NO start Address	/G スイッチが指定されたが，実行すべきプログラムがロードされていない．
? Loading Error	入力として与えられたファイルがリロケータブル•オブジェクト•ファイルではない．
? Out of Memory	プログラムをロードする十分なメモリがない．
? Command Error	コマンドがまちがっている．
? <ファイル> Not Found	指定されたファイルが存在しない．
% 2nd COMMON Larger /******/	最初に定義した COMMON ブロック/******/よりも大きな COMMON ブロックが現われた．
% Mult. Def. Global YYYYYY	外部参照記号 YYYYYY が重複して定義されている．
% Overlaying. Program(Data) Area	/D または /P スイッチにより既にロードされているデータを破壊する恐れがある．
? intersecting Program(Data) Area	プログラムとデータ領域が重なっている．
? Start symbol －シンボル －Undefined	/E または /G が指定されたが，そのスタートシンボルが定義されていない．
Origin Above (Below) Loader Memory. Move Anyway (Y or N)?	/E または /G が指定されたが，プログラムまたはデータのいずれかの領域の先頭アドレスが実行できない領域にある．Y と答えると，データまたはプログラムを移動させ処理を続行する．
? Can't Save Object File	ファイルをセーブしようとしたときにディスクエラーが発生した．
? Nothing Loaded	オブジェクトファイルがロードされていないのにライブラリの検索，リンクローダの終了処理プログラムの実行が要求された．

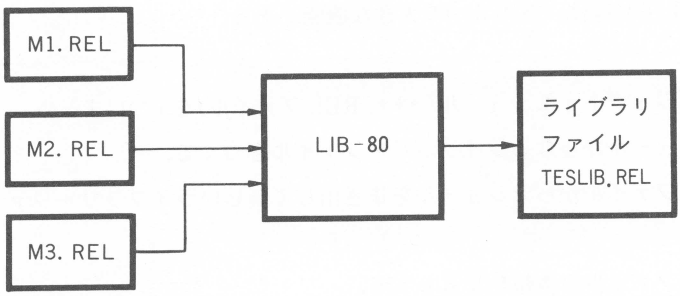
3.5 ライブラリマネジャ(LIB−80)の効果的な使用法

この節では，ライブラリマネジャとして，マイクロ・ソフト社の LIB−80 の使用法を，次の手順で解説する．

◎ LIB−80の機能

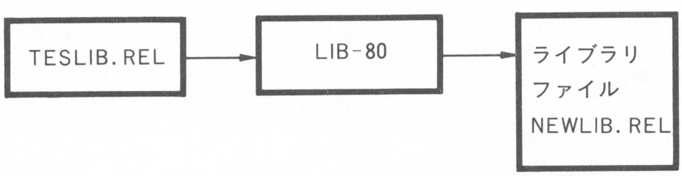


◎　オブジェクト・ファイルからのライブラリの作成例




```
A>LIB
*TESLIB=
*M1, M2, M3
*/E
A>
```

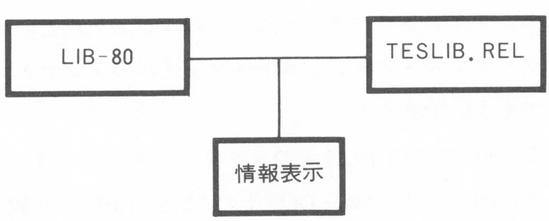
◎　ライブラリ・ファイルからのモジュールの抽出




```
A>LIB
*NEWLIB=
*TESLIB<M2>
*/E
A>
```

◎　ライブラリ・ファイルの情報の表示

```
A>LIB  TESLIB/L
```



3.5.1 ライブラリマネジャの機能

ライブラリマネジャ(LIB−80)には，次の3つの大きな機能がある．

- 個々に開発したオブジェクト・モジュール(***.RELファイル)を1つにまとめて，オブジェクト・モジュールからなるライブラリ・ファイルをつくる．
- 既存のライブラリ・ファイルからモジュールを抜き出して新しいライブラリ・ファイルをつくる．
- 既存のライブラリ・ファイルの情報をリストする．

これらの機能は，ライブラリマネジャのコマンドとスイッチの組み合わせにより選択することができる．ここで重要なことは，このライブラリマネジャが扱える最小単位であるモジュールという単位であるが，これをよく理解していないとこのライブラリマネジャを効率よく使用することができない．

ここでのモジュールという単位は，

> MACRO−80でアセンブルし出力されたオブジェクト・ファイル
> または，
> コンパイラでコンパイルされたオブジェクト・ファイル

ということになる．このモジュールという単位は，ライブラリマネジャだけでなく，リンクローダでの/Sスイッチをつけたライブラリ・ファイルからのモジュールのリンクの際に扱われる最小の単位ともなる．ライブラリマネジャは，このモジュールという単位でライブラリ・ファイルと呼ばれるリロケータブル・オブジェクト・ファイルのメンテナンスを行なうユーティリティ・プログラムと考えてもよいであろう．

ただし，このLIB−80というCP/M上のライブラリマネジャは，ライブラリ・ファイルのアップデイトということに関しては，MS−DOS上の“MS−LIB”と比較すると，はるかにその機能は貧弱で，

> 複数のモジュールを結合して，新しいライブラリ・ファイルをつくる．

という機能しかない．すなわち，既存のライブラリ・ファイルに対して，新しいモジュールを付加したり，一部のモジュールを削除するというような機能はないのである．このような処理をしたい場合には，新しくライブラリ・ファイルをつくり直すという作業になってしまう．

3.5.2　ライブラリ・ファイルの作成法

LIB－80を使用してライブラリ・ファイルを作成する場合，“LIB－80の起動の方法”，“LIB－80のコマンドのフォーマット”，“コマンド・スイッチの使用法”の3つのことを理解していればよい．

ここでは，これらのことについて解説する．

(A)　LIB－80の起動の方法

LIB－80を起動させるには2つの方法がある．この2つの方法の相違点は，LIB－80の起動コマンドである“LIB”につづいて作業内容を表わすコマンドを入力する方法と，LIB－80を起動させた後に，LIB－80が返してくるプロンプトに対してコマンドを入力する方法である．

これは，CP/Mの標準コマンドであるPIPの場合と同じようなものと考えればよい．

また，このコマンドのフォーマットは，MACRO－80のそれときわめて似通ったものとなっている．

```
方法1　A＞LIB　コマンド　（改行）
方法2　A＞LIB　　　　　　（改行）
　　　　＊コマンド　　　　（改行）
　　　　　　≀
　　　　＊/E　　　　　　　（改行）　　←　LIB－80からの脱出
```

方法1の場合のメリットは，LIB－80のコマンドラインに作業内容を表わすコマンドを記述できるので，SUBMITコマンドを使用したバッチ処理が可能となる(XSUBコマンドを使用すれば，方法2を使用してもバッチ処理を行なうことができる)．

方法2の場合のメリットは，このライブラリマネジャに用意されている各種のスイッチを使用して，よりきめの細かい作業が行なえるということにある．

これらのことより，簡単なライブラリ・ファイルの作成(アセンブラから出力されたいくつかのオブジェクト・ファイルを1つのライブラリ・ファイルとしてまとめるような場合)や，既存のライブラリ・ファイルの内容をチェックする(Lスイッチの使用)だけならば，方法1のコマンド・ライン上へのコマンドの記述による方法でよいが，一般的には，方法2を使用したほうがよい場合が多い．

ただし，この方法でライブラリマネジャを起動させた場合には，CCPにもどるときのスイッチの機能(EとRと^Cの関係)には十分に注意しないとライブラリ・ファイルが作成されずにCCPにもどってしまうことがある．

(B) コマンドのフォーマット

LIB−80のコマンドは，“ファイル仕様[/スイッチ]”が基本になっている．すなわち，ライブラリ・ファイルの作成という作業においては，まず，何というファイル名で，ライブラリ・ファイルを作成するのかを，ライブラリマネジャに知らせなければならない．これを行なわないとライブラリマネジャは，“FORLIB”という名前のライブラリ・ファイルを作成する．

ユーザがライブラリ・ファイルの名前を与えるには，他の作業に先行して次の作業を行なわなければならない(以下，*マークはライブラリマネジャ・プログラムからのシステム・プロンプトである)．

*ライブラリ・ファイル名＝(改行)

これにより，以下に指定するライブラリのためのモジュールは，このライブラリ・ファイル名で指定したファイルに登録される．このとき，ライブラリ・ファイル名に拡張子を付けてはならない．なぜならば，この時点では，ライブラリマネジャにより，

“ライブラリ・ファイル名．LIB”

という一時的なファイルとしてディスク上で扱われ，後に/Eまたは/Rスイッチにより，

“ライブラリ・ファイル名．REL”

という正式なライブラリ・ファイルに変換されるからである．

この作業以後のプロンプトに対しては，Lスイッチを使用したとき以外に，ファイル仕様を指定した場合には，このライブラリ・ファイルに登録する入力ファイルとして扱われる．ここで，入力ファイルとして既存のライブラリ・ファイルを指定した場合には，そのライブラリ・ファイル内のモジュール別の指定もできる．これを行なわない場合には，そのライブラリ・ファイルに含まれているすべてのモジュールを対象として，新しいライブラリ・ファイルに登録される(入力ファイルがライブラリ・ファイルでない場合には，そのファイル全体が1モジュールであるからモジュールの指定という概念はなくなる)．

このライブラリ・ファイル内のモジュールの指定は，次のようなフォーマットになる．

ライブラリ・ファイル名<モジュール指定>

すなわち，ライブラリ・ファイル名に続いて，目的とするモジュールをカギ括弧<>でくくって指定するというフォーマットとなっている．

このカギ括弧でくくったモジュールの指定の方法も次の3通りの方法がある．

a) モジュール名を直接指定する．
b) あるモジュールからの±255の相対的な位置で指定する．
c) 2つの範囲で指定する．

a)の方法は最も簡単なもので，例えば，OLDLIBというライブラリ・ファイルの中のMOJ5というモジュールを指定する場合には，

*OLDLIB<MOJ5>

とすればよい．

b)の方法は，あるモジュールの前後の何番目かのモジュールを相対的な位置で指定するという方法で，例えば，MOJ5の1つ前のモジュールを指定したいというときには，

*OLDLIB<MOJ5−1>

とすればよい．

c)の方法は，一度に複数のモジュールを範囲を与えることによって指定できるもので「〜から〜まで」という言葉に対応する記号として“..”という記号を使用する．例えば，「MOJ3からMOJ7まで」というような場合，

*OLDLIB<MOJ3..MOJ7>

という指定を行なう．この“..”の前が省略された場合，つまり<..MOJ7>というような指定の場合には，最初のモジュールからMOJ7までを指定したことになる．

この範囲の指定には，b)の相対的な指定の方法を組み合わせてもよい．すなわち，

*OLDLIB<MOJ3−1..MOJ9+1>

とした場合には，MOJ3の1つ前のモジュールから，MOJ9の1つ後ろのモジュールまでを指定したことになる．

また，このカギ括弧の中にa)からc)までの組み合わせを使用して複数のモジュールを指定することもできる．その場合，それぞれのモジュールは，カンマ“,”で区切らなければならない．

例えば，

*OLDLIB<MOJ3..MOJ8,MOJ17>

というようなフォーマットになる．

この入力ファイルの指定は，プロンプトに対して1ファイルずつ答えてもよいし，カンマで区切って複数のファイルを一度に指定してもよい．

*SUB1,OLDLIB<MOJ1>,SUB5

(C) コマンド・スイッチの使用法

このライブラリマネジャのコマンド・スイッチというのは，ライブラリ・ファイルを作成するために使用するものと，既存のライブラリ・ファイルをチェックするときに使用するものとに分けて考えることができる．また，これらのスイッチで，コマンドと組み合わせて使用しなければならないものは，Lスイッチと Uスイッチだけで，それ以外のものはライブラリマネジャが返してくるプロンプトに対して単独で使用できる．また，これらのスイッチの前には必ずスラッシュ“/”を付けなければならない．

以下に，これらのスイッチの機能をまとめる．

◎ ライブラリ・ファイル作成に関するもの

スイッチ	機 能
C	それまでの作業を破棄して初期状態にもどす.
E (End)	作業に使用していた"***.LIB ファイル"を正式なライブラリ・ファイル"***.REL ファイル"としてディスクに登録しCCPにもどる.
R (Rename)	Eと同じであるがディスクに登録後CCPにもどらず，連続して作業を続けることができる. ただし，この状態でCCPにもどるときには，Eスイッチを使用せずに^C(コントロールC)でもどらなければならない.

◎ ライブラリ・ファイルのチェックに使用するもの

スイッチ	機 能
L (List)	ライブラリ・ファイル名/Lで，そのライブラリ・ファイルのモジュールに関する情報を表示する (^Pをセットすることによりプリンタへの出力となる. 次のUスイッチの場合も同様).
U (Ufo)	ライブラリ・ファイル名/Uで，そのライブラリ・ファイル内の未定義の記号を表示する.
O	Lスイッチに対する表示を8進数で行なう.
H	Lスイッチに対する表示を16進数で行なう. デフォルトではこのモードである.

3.5.3 ライブラリ化の実例

ここでは，"4.2.5(A)アセンブルの実例"で示した乗算と除算のモジュールをライブラリ化する方法を示す. 作成するライブラリのファイル名は"SUB.REL"とする.

≪ 操 作 ≫

A>LIB (改行)

*SUB=M16,D16(改行)

(D16 ← ライブラリに登録するリロケータブル・ファイル(D16.REL))

(M16 ← ライブラリに登録するリロケータブル・ファイル(M16.REL))

(SUB ← 作成するライブラリの名前(SUB.REL))

[この時点で，ディスク上に"SUB.LIB"というファイルがつくられる.]

*/E (改行) ←SUB.LIBが，SUB.RELに変更され，CCPに戻る.

[この時点で，ディスク上に"SUB.REL"というファイルが作成される.]

A>░

次にこのライブラリ・ファイルの内容をチェックする.

A>LIB (改行)

*SUB/L (改行) ←ライブラリSUB.RELの情報を表示する.

```
Module M16 of SUB REL      ←モジュールのタイトル
Length of Program   22     ←モジュールM16のプログラム長は22バイトである.
Length of Data area  0     ←モジュールM16のデータ長は0バイトである.
Entry point(s):
M16    0000'        ←D16の開始アドレス(相対値)
Module D16    of SUB    REL     ←モジュールのタイトル
Length of Program          45  ←モジュールD16のプログラム長は45バイトである.
Length of Data area         0  ←モジュールD16のデータ長は0バイトである.
Entry point(s):
D16    0000'        ←D16の開始アドレス(相対値)
Symbol  Value  Defined  Referenced
M16     0000'  M16
D16     0000'  D16
 ↑       ↑      ↑
 │       │      └──── ライブラリ内で区別できるモジュール名
 │       └─────────── エントリーアドレス
 └─────────────────── 使用されているシンボル名
*^C
A>
```

3.6　シンボリック・デバッガ(ZSID)の効果的な使用法

この節では，シンボリック・デバッガとして，ディジタルリサーチ社のZSIDの使用法を，次の手順で解説する．

3.6.1　シンボリック・デバッガの機能
3.6.2　シンボリック・デバッグの方法
3.6.3　シンボリック・デバッグの実例

ZSIDの機能

デバッグ用ファイル
***.COM
(***.HEX)

シンボリックデバッグの場合

シンボルファイル
***.SYM

ZSID(SID)
コマンド

1.	1ライン・アセンブラ	A(Assemble)
2.	サブルーチン・コール	C(Call)
3.	ディスプレイ・メモリ	D(Display memory)
4.	メモリ充填	F(Fill memory)
5.	テスト・プログラム実行	G(Go to program)
6.	16進数計算および表示	H(Hex values)
7.	入力ファイル名設定	I(Input line)
8.	逆アセンブラ	L(List code)
9.	メモリ内容の移動	M(Move memory)
10.	ブレイク・ポイント設定	P(Pass counter)
11.	ファイル読み込み	R(Read files)
12.	メモリ書き込み	S(Set memory)
13.	トレース実行	T(Trace mode)
14.	アントレース実行	U(Untrace mode)
15.	レジスタ状態表示	X(eXamine cpu state)

3.6.1　シンボリック・デバッガの機能

シンボリック・デバッガとは，機能的にはCP/Mに標準でバンドリングされている“DDT”と同じデバッグ・ツールであるが，DDTと異なるところは，アドレスの表現などにシンボル（名前）

が使用できるということである．ただし，この**シンボリック・デバッグを行なうためには，必ずシンボル・ファイル“＊＊＊．SYM”というファイルが必要**になる．このファイルがない場合には，単なるDDTと同じような機能しかない．

それゆえ，“ASM(CP/Mにバンドリングされているアセンブラ)”のように，シンボル・ファイルを出力しないアセンブラでは，このシンボリック・デバッガを使用することはできない．

これらのシンボル・ファイルを出力するアセンブラとしては，ディジタル・リサーチ社の”MAC，RMAC”，マクロ・ソフト社の“MACRO−80”などがある．また，このシンボリック・デバッガにもいくつかの製品があり，それぞれ特長があるが一般的なものとしては，ディジタル・リサーチ社の“SID，ZSID”，ライフボート社の“S−BUG”などがある．これらの中で機能的に論じるならば，後発の“S−BUG”がコマンドの種類などからも優れているが，これらのデバッガの基本になっているのがディジタル・リサーチ社の“SID，ZSID”であり，CP/M上のシンボリック・デバッガのスタンダード的な存在であるので，ここでは，このディジタル・リサーチ社のデバッガについてその機能を論ずることにする(S−BUGには，このディジタル・リサーチ社のデバッガのコマンドは，すべて含まれているのでそのまま使用することができる．

この“SID”と“ZSID”の相違は，前者がインテル・ニーモニックによるi8080用のデバッガであるのに対して，後者は，ザイログ・ニーモニックによるZ80用のデバッガであるということを除けば，全く同じ機能をもつものである．現にこれらのマニュアルは，一部を除いて共通になっている．

ここでは，解説してきたアセンブラがMACRO−80であり，対象とするCPUがZ80であることから，ZSIDを使用しての解説になっているが，i8080とZ80のニーモニックの相違とZ80の拡張された命令の部分を除けば，全く同じ機能であることに注意してほしい．

このZSIDの機能は，大別すると2つに分けて考えることができる．それは，

- Z80用のDDT
- Z80用のシンボリック・デバッガ

という分け方である．一般的に使用されるのは，前者の“Z80用のDDT”としての使用法であろう．著者も簡単なプログラムのデバッグのときには，もっぱら前者の“Z80用のDDT”としての使い方をしている．その理由としては，シンボリック・デバッグを行なうためには，シンボル・ファイルが必要になるわけであるが，これを出力するためにはリンク時に“/Yスイッチ”を付けなければならない．このような付加作業を行なえば当然，リンクに要する時間は長くなる．

また，ディスク上のスペースも必要になる．もっとも，これらは，たいした問題ではないのであるが，要はこのシンボリック・デバッグということがあまり効果をもたない場合が多いのである．というのは，シンボリック・デバッグといっても，

シンボルとして使用できるのは，共通参照記号(PUBLIC宣言をした名前)だけ.

なのである．これはこのシンボリック・デバッグの対象となるプログラムがリンクローダを使用して複数のモジュールを結合したものであれば，全くあたりまえのことなのであるが，他のモジュールをリンクしないスタンドアローンのプログラムの場合には，当然“PUBLIC宣言をした名前”などというものはないわけで，すなわち，空のシンボル・ファイルしか作成されず，全く意味のないシンボル・ファイルになってしまい，シンボリック・デバッグができないことになる．これがシンボリック・デバッグを行なわなくなってしまう原因である．であるから，

シンボリック・デバッグとは，モジュール開発を行なったプログラムを対象に行なう.

と覚えておけばよいであろう．先にも述べたようにシンボリック・デバッグを行なえば，実アドレスの代わりに名前を使用できるのできわめて便利なこともある．そして，このシンボリック・デバッガでは，そのコマンドのすべてに，この名前による指定ができるようになっているので実アドレスの入力ミスというような初歩的なミスを防ぐこともできる．

3.6.2　シンボリック・デバッグの方法

ZSIDを使用してデバッグを行なう場合には，3.6.1で解説したように，単に“Z80用のDDT”としての使用法と，シンボル・ファイルを使用してシンボリック・デバッグを行なう方法があるが，基本的には同じものであるので，ここでは，“ZSIDの起動の方法”，“ZSIDの基本的なルール”，“ZSIDのコマンドの使用法”を解説し，次節でシンボリック・デバッグの実例を示す．

(A)　ZSIDの起動の方法

ZSIDを起動させる方法は，DDTのそれとなんらかわることはない．ただし，このZSIDの場合には，シンボリック・デバッグを行なうためのシンボル・ファイルの指定ということがあるために，次の5通りの方法がある．

(A)	ZSID(改行)		
(B)	ZSID ***.COM(改行)		
(C)	ZSID ***.HEX(改行)		
(D)	ZSID ***.COM	***.SYM(改行)	シンボリック
(E)	ZSID ***.HEX	***.SYM(改行)	デバッグの場合

すなわち，単にZSIDを起動させるだけならば，ZSIDとタイプして改行するだけでよい．ただし，この方法でZSIDを起動させた場合にはI(Input)コマンドとR(Read)コマンドを使用してデバッグの対象となるファイルをロードしなければならない．これもまたDDTの場合と同じである．基本的にデバッグの対象となるファイルは，実行形式ファイル(***.COM)か，インテルHEXフォーマット・ファイル(***.HEX)でなければならない．MACRO−80システムの場合にはリンクローダの標準の出力が実行形式ファイルであるから，これを使用するのが普通である．ASMからの出力ファイルはHEXファイルであるから，ASMシステムでシンボリックデバッガを使用する場合には，これを使用したほうがよい(注．ASMの場合，インテル・ニーモニックであるからZSIDでなくSIDを使用する)．

シンボリック・デバッグを行なうためには，さらにシンボル・ファイルを指定しなければならない．このシンボル・ファイルの拡張子は“.SYM”でなければならない．これを指定する場合には，デバッグの対象となるファイルに続いて拡張子までを含めた形のファイル名を指定する．これは，ZSIDの内部でIコマンドとRコマンドを使用する場合でも同様である．

(B)　ZSIDの基本的なルール

ZSIDの内部で利用できる文字やコントロール・コードは，基方的にはDDTのそれとほとんどかわるところはないが，次の点において拡張が見られる．

1. 10進数が使用できる．
   ZSIDの内部で扱う数値は一般的には16進数であるが，10進数で表現したいときには，その数値の頭に“#”マークを付けることにより，その数値は10進数として扱われる．
   例：　#225，#9999
2. ASCII文字を扱うことができる．
   アポストロフィ(')でくくることによって，ASCII文字を直接使用することができる．
   例：　F100, 200, 'A'

またシンボリック・デバッグを行なった場合にそこに示されるシンボルには次のルールがある．

1. アドレスを表わすシンボルの場合，そのシンボルの前にピリオド(.)が付く．
   例：　.START
2. 16ビットの値(データ)を表わすシンボルの場合，そのシンボルの前に“@”マークが付く．　例：　@WORD
3. 8ビットの値(データ)を表わすシンボルの場合，そのシンボルの前にイコール(=)記号が付く．　例：　=BYTE

これらのシンボルに関するルールは，コマンドのオペランドとして使用するときにも，守らなければならない.

すなわち，G(GO)やC(Call)の場合にシンボルによる指定を行なう場合は，必ず，目的とするシンボル名の頭にピリオド(.)を付けなければならない.

(C) コマンドの使用法

ZSIDのコマンド(SIDも同じ)は，DDTのコマンドをすべて含んでおりそのコマンド名は，DDTのそれと全く同じものとなっている．しかし，同じコマンド名であってもオプションの機能が強化されていたり，フォーマットが微妙に異なるものがある．これらの変更点は，いずれもこれまでのDDTで不満であったものが改良されているといってよいであろう.

それ以外にZSIDでは，次の2つのコマンドが新設され非常に使用しやすくなっている.

C(Call subroutine)：サブルーチンのコール
P(Pass counter)　：通過回数を指定した恒久的ブレイク・ポインタの設定

この2つの新設されたコマンドは，これまでDDTを使用して，いつも不便を感じていたものであり，特にPコマンドの新設によりサブルーチン内のデバッグや，ブレイクポイントを忘れてGコマンドを実行してしまい，プログラムを暴走させるというようなケアレス・ミスが大幅に減少した.

また，Cコマンドは，テストスタブをサブルーチンの形で用意しておいて，これによりテスト・プログラムにデータを与えるような場合，単独にこのテストスタブを実行させることができるのできわめて重宝している．また，これらのコマンドの中には，ユーティリティ・プログラムとして用意されている，

HIST．UTL(Histgram Utility)
TRACE．UTL(Trace Utility)

のためのものもいくつかあるが，著者は，これらのユーティリティをマニュアルの不備とマニュアルの解説からは，あまり利用価値を認められないところから，これらを使用したことがない．そこで，これらに関連するものは省略してこのZSIDのコマンドの機能をまとめてある．DDTに比較して機能が拡張されたもの，および，改良されたものにはアスタリスク(*)が付してあるので注意してもらいたい.

なお，実際のコマンドの使用例に関しては，“3.6.3 シンボリック・デバッグの実例”を参照してほしい.

[ZSID のコマンド]

A(Assemble)	ザイログ形式のニーモニックによる1ライン・アセンブラ
A	現在のプログラム・カウンタがさすアドレスからのアセンブル.
Assss	アドレス ssss からのアセンブル.
* −A	アセンブラと逆アセンブラ(L コマンド)の機能を取り除き，その分のメモリを開放する.
* C(Call)	サブルーチンの実行(コール)
Cssss	アドレス ssss からのサブルーチンを実行する.
Cssss, xxxx	BC レジスタに xxxx を，DE レジスタに 0000 を設定し，アドレス ssss からのサブルーチンを実行する.
Cssss, xxxx, yyyy	BC レジスタに xxxx を，DE レジスタに yyyy を設定し，アドレス ssss からのサブルーチンを実行する.
D(Display memory)	メモリ内容の表示
D	現在の表示アドレスの続きから，1/2 画面分のメモリの内容を，16 進数と ASCII 文字で表示する.
Dssss	アドレス ssss から上記と同じ機能をする.
Dssss, eeee	アドレス ssss から eeee までを表示する.
D, eeee	現在の表示アドレスから eeee までを表示する.
F(Fill memory)	メモリへのデータの設定
Fssss, eeee, xx	アドレス ssss から，eeee のメモリにデータ xx を書き込む.
G(Go to program)	プログラムの実行
G	現在のプログラム・カウンタが示すアドレスからのプログラムの実行.
G, qqqq	現在のプログラム・カウンタが示すアドレスからプログラムを実行し，アドレス qqqq で停止させる.
G, qqqq, zzzz	現在のプログラム・カウンタが示すアドレスからプログラムを実行し，アドレス qqqq または zzzz で停止させる.
Gssss	アドレス ssss からプログラムの実行.
Gssss, qqqq	アドレス ssss から実行し，アドレス qqqq で停止させる.
Gssss, qqqq, zzzz	アドレス ssss から実行し，アドレス qqqq または zzzz で停止させる.
* −G. . .	この形は G に続くものとして，これまでのいずれの形をとることもできるが，P コマンドによるパス・ポイントを，パス・カウンタが 0 になるまでトレースしないというところが異なる.
H(Hex values)	16 進数の計算およびシンボルの値の表示
Hxxxx, yyyy	xxxx と yyyy の和(xxxx＋yyyy)と差(xxxx−yyyy)を 16 進数で表示する.
* Hxxxx	xxxx の値を 16 進数で表示する.
* H	使用されているすべてのシンボルの値を 16 進数で表示する.

＊ I(Input line)	R コマンドで読み込むファイルの名前を設定する．複数のファイルを指定してもよいし，またドライブ番号も指定できる． 例：I B:T.COM　T.SYM
L(List code)	逆アセンブラ
L	現在のリスト・アドレスから，画面の 1/2 分のステップ数逆アセンブルする．
Lssss	アドレス ssss から，画面の 1/2 分のステップ数逆アセンブルする．
Lssss, eeee	アドレス ssss から，eeee まで逆アセンブルする．
＊ －L．．	上記3種類の使用法ができるが，シンボルを表示しない．
M(Move memory)	メモリ内容のブロック移動
Mssss, eeee, dddd	アドレス ssss から，eeee までのメモリの内容をアドレス dddd からに移動する．
＊ P(Pass counter)	ブレイク・ポイントの設定および表示
Pssss	アドレス ssss に通過回数1のブレイク・ポイントを設定する．
Pssss, xx	アドレス ssss に通過回数 xx のブレイク・ポイントを設定する．
P	現在の P コマンドの使用状況を表示する(ブレイク・ポイントを設定したアドレスと，その回数を表示する)．
－Pssss	アドレス ssss のブレイク・ポイントを解除する．
－P	すべてのブレイク・ポイントを解除する．
R(Read file)	I コマンドで設定したファイルをメモリに読み込む．
S(Set memory)	メモリにデータを書き込む
Sssss	アドレス ssss からバイト単位でデータを書き込む．
＊ SWssss	アドレス ssss からワード(2バイト)単位でデータを書き込む．
T(Trace mode)	トレース・モードでプログラムをステップ実行する．
T	1ステップ，トレース・モードで実行する．
Txx	xx ステップ，トレース・モードで実行する．
＊ －T．．	1, 2 のいずれの形でも実行できるが，シンボルを表示しない．
＊ TW．．	1, 2 のいずれの形でも実行できるが，サブルーチンに関しては，トレースしない．
＊ －TW．．	上記の形でシンボルを表示しない．
U(Untrace mode)	アントレース・モードでプログラムをステップ実行する．
U	1ステップ，アントレース・モードで実行する．
Uxx	xx ステップ，アントレース・モードで実行する．
＊ －U．．	1, 2 のいずれの形でも実行できるが，シンボルを表示しない．
＊ UW．．	1, 2 のいずれの形でも実行できるが，サブルーチンに関しては，アントレースしない．

＊ －UW.	上記の形でシンボルを表示しない.
X(eXamine)	レジスタ状態の表示と変更
X Xreg	現在のレジスタ類の状態を表示する. reg で示したレジスタ類の状態(内容)を変更する.

注意）・アドレス表現は，16 進数，# を付けた 10 進数または，ピリオドを伴うシンボルとなる.
　　例： 1000　#130　.START
・ワード•データは，16 進数，# を付けた 10 進数または，@ マークを伴うシンボルとなる.
　　例： FFFF　#1024　@ WORD. DATA
・バイト•データは，16 進数，# を付けた 10 進数，アポストロフィ(')でくくられた ASCII 文字または，＝マークを伴うシンボルとなる.
　　例： FA　#128　'T'　= BYTE. DATA

3.6.3　シンボリック•デバッグの実例

"3.4.3(A) リンクの実例" で作成した "MAIN. COM" と "MAIN. SYM" を使用して ZSID によるシンボリック•デバッグを行なう.

```
A>ZSID  MAIN.COM  MAIN.SYM
               ↑          ↑
               │          └─シンボル・ファイルの指定
               └────────────実行ファイルの指定
ZSID  VERS  1.4
SYMBOLS  ← シンボリック•デバッグであることを示す
NEXT  PC   END
0180  0100  A9E4
 ↑     ↑     ↑
 │     │     └──メモリの最終アドレス
 │     └────────ロード開始アドレス
 └──────────────ロード最終アドレス
#L100,157←100 番地から 157番地まで逆アセンブル
  0100  JP    0103 .START ←┬共通参照記号の部分はシンボルが表示される
START:  ←──────────────────┤
  0103  LD    HL,000A      │
  0106  LD    DE,0002      │
  0109  CALL  0115 .M16 ←──┤
  010C  LD    DE,0004      │
```

```
  010F  CALL   012B.D16  ←
EXIT: ←
  0112  JP     0000
M16: ←
  0115  LD     C,L
         ≀
  012A  RET
D16: ←
  012B  LD     A,E
         ≀
  0157  RET
```

外部モジュールも完全にリンクされている

```
 #P109     ←  109 番地にパスポイント設定
 #P10C     ←  10C 番地にパスポイント設定
 #P.EXIT   ←  EXIT 番地にパスポイント設定
              (シンボルがアドレスの場合には，ピリオドをシンボル名の前に付ける)
 #P    ←パスポイントの確認
 01  0109
 01  010C
 01  0112   .EXIT ←シンボルが定義されているところは，それが示される
 ↑    ↑
 |    └──ブレイク・アドレス
 └────────パス回数
#UFF  ←第1パスポイントまでの実行
 -----  A=00 B=0000 D=0000 H=0000 S=0100 P=0100
 -----  A' 00 B'0000 D' 0000 H' 0000 X=0000 Y=0000 JP 0103 .START
01 PASS 0109 ←第1パスポイントでブレイク
 -----  A=00 B=0000 D=0002 H=000A S=0100 P=0109
 -----  A'00 B'0000 D' 0000 H' 0000 X=0000 Y=0000 CALL 0115 .M16
*0115 .M16  [DE と HL レジスタは，指定したとおりの値が設定されている]
#UFF  ←続いて第2パスポイントまで実行
 -----  A=00 B=0000 D=0002 H=000A S=00FE P=0115
 -----  A'00 B'0000 D' 0000 H' 0000 X=0000 Y=0000 LD  C,L
01 PASS 010C ←第2パスポイントでブレイク(M16 の実行)
```

```
 -Z-E-  A=00  B=000A  D=0000  H=0014  S=0100  P=010C
 -----  A'00  B'0000  D'0000  H'0000  X=0000  Y=0000  LD  DE,0004
*010F          [HLレジスタの値は正しく 2倍されている  2×AH=14H]
#UFF  ←続いて第3パスポイントまで実行(D16の実行)
 -Z-E-  A=00  B=000A  D=0004  H=0014  S=0100  P=010F
 -----  A'00  B'0000  D'0000  H'0000  X=0000  Y=0000 CALL  012B .D16
01 PASS 0112 .EXIT ←第3パスポイントでブレイク
 -Z-E-  A=00  B=0005  D=0000  H=0005  S=0100  P=0112
 -----  A'00  B'0000  D'0000  H'0000  X=0000  Y=0000  JP   0000
*0000          [HLレジスタの値は正しく1/4になっている  14H/4=5]
```

◇プログラムは正常に動作している．・・・デバッグ終了◇

注意）この問題ではスタックポインタが設定してないが，実際のプログラムではこれを最初に設定しなければならない．

参考文献

1. 押野崇芳『マイコンを使いこなすための“アセンブラ入門”』日刊工業新聞社，1983
2. 前田英明『詳解 CP/M の使い方』工学図書株式会社，1981
3. 村瀬康治『応用 CP/M』アスキー出版局，1982
4. Alan R. Miller 著，友枝英一 訳『マスタリング CP/M』CQ 出版社，1984
5. 前田英明『マクロ・アセンブラの使い方』工学図書株式会社，1984
6. 菅野孝男『ソフトウェア開発のマネージメント』廣済堂産報出版，1984

索　引

（五十音順）

[わ　行]

[欧　文]

[その他]

●著者紹介●

押野崇芳（おしの たかよし）

昭和28年４月　山梨県に生まれる
昭和51年３月　東京電機大学電気通信工学科卒業
昭和 52 年　同大学通信第一研究室にてレーザー通信の研究に従事
昭和52～56年　日本電子専門学校にてマイコン教育およびマイコン教育機材の開発・研究に従事
現在　Education Adviser
東京電機大学非常勤講師，㈱ヴァル研究所非常勤研究員，日本能率協会講師

主な著書に「マイコンを使いこなすためのアセンブラ入門」(日刊工業新聞社)，「ＰＣ8801　ＢＡＳＩＣ＋機械語プログラムの構造化手法入門」(日刊工業新聞社)，「ＰＣ8801　ＢＡＳＩＣ＋機械化プログラムによるディスク活用技法」(日刊工業新聞社)，8086 16ビットCPUアセンブラ入門（日刊工業新聞社）

〔Office〕〒166　東京都杉並区阿佐谷北2-36-1
キングダム阿佐谷204
TEL(03)337-9419

CP/MによるZ80マクロ・アセンブラ入門　　NDC 549.92

昭和59年10月31日　初版１刷発行
昭和62年５月29日　初版６刷発行

発行者　藤吉敏生
発行所　日刊工業新聞社
東京都千代田区九段北一丁目８番10号
（郵便番号102）
電　話　東京263-2311（大代表）
振替口座　東　京　９－186076

印刷・製本　㈱コーラル　（組版：南西印刷　印刷：萬友社　製本：養正社）

落丁，乱丁本はお取り替えいたします。　（定価はカバーに表示してあります）

ISBN4-526-01787-6 C3034